MERIDIAN
SOFTWARE SYSTEMS

Meridian Ada 4.1

Compiler User's Guide
PC DOS

Summary of Changes

The "Summary of Changes" section documents changes made to the compiler for this new release. If you are a first time user of the Meridian Ada compiler, this section may not be of interest to you.

If you are were a previous user of Meridian compilers, you will notice some significant changes in the new version. The most obvious change being the name. The compiler has been renamed from *Meridian AdaVantage* to *Meridian Ada*.

The *Compiler User's Guide* has undergone a complete restructure. It is now broken into two parts with Part I, Using the Compiler, documenting how to use the compiler. Part II, Meridian Ada, contains an alphabetical listing of all Meridian Ada commands.

The Installation instructions have been moved to the *Getting Started* manual.

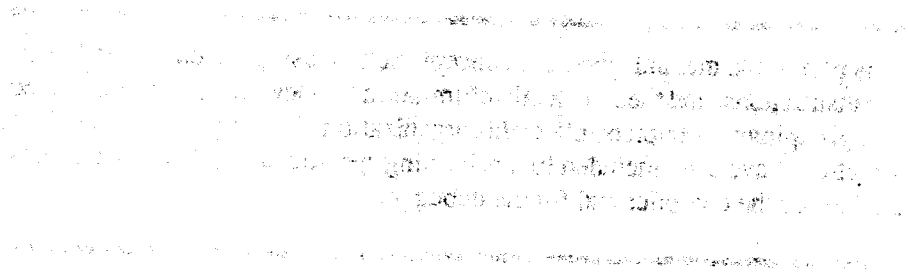
The following updates and new features have been added to the Meridian Ada compiler for release 4.1:

- Signed ranges of integers can now be packed.
- The DOS print screen function can be issued from an executing Ada Program.
- Validated under ACVC version 1.11.

Table of Contents

Part I Using the Compiler

This part of the manual gives the concepts behind using the compiler, basic how to use instructions, and then in–depth information for Developers and Advanced users. The groupings of chapters reflect this organization. Where possible, step by step instructions have been included for performing specific tasks. Tutorials have been included for the compiler and for the debugger.



Chapter 1 Before You Begin

1.1 About Meridian Ada

Meridian Ada™ compilers provide fast, efficient, complete, low priced, production quality Ada programming capabilities for the IBM PC™, Apple Macintosh™, DECstation™, Stardent Titan™, Sun 3™ and Sun 4™ computers among others. Contact your sales representative for a complete list of supported hardware.

Full generics, tasking, and separate compilation are supported. All standard packages are provided. All of the applicable implementation–dependent system programming features (the *Reference Manual for the Ada Programming Language** ANSI/MIL–STD–1815A Chapter 13) are implemented. A set of library management tools provides control over programming project organization and databases of compilation units. Auxiliary directory arrangements permit separation of source code and compiler generated files.

This *User's Guide* covers both the standard DOS version of Meridian Ada and the Extended Mode version. The Extended Mode version of Meridian Ada must be purchased separately. The Extended Mode version of Meridian Ada allows you to create DOS programs up to 16MB.

The compiler comes with the following associated components:

- The Meridian Ada Debugger, an interactive source–level debugger for use with programs written using the Meridian Ada compiler.
- The Meridian Ada Utility Library, a set of Ada packages for use with the compiler.
- The Meridian Ada DOS Environment Library, an Ada component library and productivity enhancing tool.
- The Meridian Ada Optimizer, a facility that performs a variety of local and global optimizations.

1.2 Road Map for this Manual

This manual is a combination *User's Guide* and *Reference Manual*. This manual has been divided into several sections.

"Part I Using the Compiler" gives the concepts behind using the compiler, basic how to use instructions, and then in–depth information for Developers and Advanced users.

The Reference portion of the manual, "Part II Meridian Ada Command Reference", lists all the Meridian Ada commands in alphabetical order.

1.3 What You Should Know

It is assumed that you are familiar with the Ada language as described in the LRM. If you are not familiar with Ada, please refer to the many excellent tutorial and reference works available.

*The *Reference Manual for the Ada Programming Language* ANSI/MIL–STD–1815A is more commonly referred to as the LRM. To enhance readability, all future references to the Reference Manual for the Ada Programming Language (ANSI/MIL–STD–1815A) are shortened to the LRM.

It is further assumed that you have some familiarity with the PC–DOS or MS–DOS ™ operating systems, including conventions for naming files and invoking programs. You should also have working knowledge of a text editor.

1.4 Integration with ACE

The Ada Compiler Environment (ACE) is a menu driven, multi–window interface with tightly coupled integration to the Meridian Ada compilation system. ACE also contains a powerful text editor with Ada template editing capability, an online Language Reference Manual, extensive online help, and interfaces to other Meridian tools.

Meridian Ada can be used with ACE or directly from the DOS prompt. This *User's Guide* describes the commands as they would be used from the DOS prompt. You will need to refer to this manual for technical details about the compiler. Refer to the *Meridian ACE User's Guide* for instructions on how to use the compiler with ACE.

1.5 Scope of This Document

This document describes operation of the Meridian Ada compiler, the associated library management facilities, and implementation dependencies. Both the standard and extended mode versions of Meridian Ada are covered in this guide.

The documentation for the Meridian Ada Utility Library, a supplemental utility library, is included in this manual. This is a collection of utility packages that provide access to command line arguments, perform bit manipulations and transcendental math functions, and provide more advanced text handling capabilities.

The Meridian Ada DOS Environment Library is a system interface package that allows you to use DOS system functions, to call BIOS functions, and to perform screen management operations. The Meridian Ada DOS Environment Library is documented separately.

Chapter 2 Using the Compiler

This chapter provides you with the basic information for using the compiler. An exercise is included in which you enter a short program and then use the Meridian Ada compiler to compile the program.

2.1 About the Compilation Process

Four basic steps are involved in taking a simple programming project from start to finish using the Meridian Ada Compiler:

1. Enter the source code.
2. Create program library.
3. Compile using the **ada** command.
4. Link using the **bamp** command.

2.2 Creating a Program

2.2.1 Entering Source Code

You can enter your program using any editor, provided that editor introduces no illegal characters into the file. (Some word processing editors embed control characters or other information that is not permitted in Ada source code files.) You should enter the program into a file in your current working directory.

The name of the Ada source code file should have the extension **.ada**, as in **prg_name.ada**.

Extensions other than **.ada** are permitted. Refer to section 19.1 for more information.

2.2.2 Ada Program Guidelines

The main subprogram must be a parameterless procedure. In accordance with the Ada language definition*, the top level of an Ada program, the *main program*, must be a subprogram that is a library unit (a top–level, non–nested compilation unit). Do not use a package (even packages that are library units) or a task as the main program.

The **bamp** command generates the code necessary to elaborate all of the library units associated directly or indirectly with the main subprogram. Elaboration of these library units is completed before the code in the main subprogram is executed. The **bamp** command is discussed in sections 2.2.5 and 19.7.

If it is necessary for a program to obtain information from the execution environment, (command line arguments for example) then a separate package designed for that purpose must be used. Section 18.1 describes a package, **arg**, for precisely that purpose.

2.2.3 Creating the Program Library

Most programs use standard library units such as package **text_io**. The program library tells the compiler where to look for **text_io** which is available in the standard library, **paclib\ada.lib**.

*See the LRM § 10.1, paragraph 8 (LRM 10.1/8).

To create a program library that references the standard library database file, use the **newlib** command. To use **newlib**, type **newlib** and press ENTER. This creates the program library file, **ada.lib**, and enters a link to **paclib\ada.lib**.

The **newlib** command simply invokes the **mklib** command (and a few other commands) with some useful defaults.

The **newlib** command should be invoked exactly once in each directory in which Ada compilations are to take place.

Additional details about the library system are given in Chapter 8.

2.2.4 Invoking the Compiler

After you have created your program, you will want to compile it. To invoke the compiler, use the **ada** command. From the command line type:

```
ada prg_name.ada
```

The compiler prints status information to your terminal. This information includes the number of lines compiled and whether or not there were any errors.

The example shown above is a simple example. The **ada** command does allow you to request additional actions. These actions are requested using command line options. Examples of command line options and their actions are **-fD**, which tells the compiler to generate debugging information and **-fE**, which generates an error file.

2.2.5 Invoking the Linker

The function of the linker is to produce an executable program by putting together the separate components of the program.

An example of separate components would be a program which uses the package **text_io**. Two visible components of the program are the package **text_io** and the program itself. Two invisible components of the program that might also be bound in are the Meridian Ada run–time code modules and other library units referenced by **text_io**.

Given that the various library database files have been properly set up, the linker need only be told the name of the main program, as in this example:

```
bamp prg_name
```

The linker finds all the components of the program, whether visible or invisible, and binds them together to produce the executable program.

2.3 Additional Information

2.3.1 Running a Program

To execute the program, simply type in the program name and press ENTER.

2.3.2 Command Line Options

All Meridian Ada commands, including the **ada** and **bamp** commands, have command line options. A command line option is a directive to a Meridian Ada command that fine tunes its operation. Command line options look like **-K** or **-fD** (single letters or groups of letters preceded with a dash). Without any command line options (sometimes called flags, arguments, switches, parameters, or just options), the Meridian Ada commands operate with default assumptions.

Some of the more useful and commonly applied command line options involve such things as invoking the Meridian Ada Optimizer, requesting more information from library listings or creating compilation listings.

Command line options are case sensitive; that is, `-l` means something different than `-L`.

Details of Meridian Ada commands and their options, along with examples, are given in Chapter 19 of this manual.

2.3.3 Kinds of Programs

There are several kinds of programs that can be created under DOS. The program types that are relevant to Meridian Ada are:

Real Mode Programs	The majority of programs that run under DOS are Real Mode programs. Real Mode programs run on any DOS system (barring any peculiar system dependencies, of course). Meridian Ada produces Real Mode programs by default.
Extended Mode Programs	Extended Mode Meridian Ada programs can be much larger than Real Mode programs (up to 16MB) and can take advantage of extended memory. Extended Mode programs can run on DOS systems with 80286 or 80386 processors. Chapter 11 discusses Extended Mode programs in detail. This feature is not included in all versions of the Meridian Ada compiler.

2.3.4 Reserved Compilation Unit Names

A list of library units in the standard distribution library follows. Although it is permitted to use these names for your own library units, you should refrain from doing so, because a "local" library unit (one compiled in your current working directory) with the same name as a "remote" library unit (such as any of those in the standard distribution library) supersedes the remote compilation unit. The compiler prints a warning message when this happens. The remote library unit is not destroyed; it only becomes locally inaccessible. This is a feature primarily of interest to those who plan to make use of the optional Run–Time Customization Library.

Package `ada_io`
Package `calendar`
Package `dcd_runtime`
Generic Package `direct_io`
Package `enum_io_runtime`
Package `file_manage`
Package `fio`
Package `fixed_io_runtime`
Package `float_io_runtime`
Package `iio`
Package `int_io_runtime`
Package `io_exceptions`
Package `machine_code`
Package `num_io_runtime`
Generic Package `sequential_io`
Package `syerr`
Package `syio`
Package `system`
Package `sytime`
Package `task_control`
Package `tcd_runtime`
Package `terminate_runtime`

Package `text_io`
Package `tioc_runtime`

2.4 Helpful Hints

This section discusses ways to make using Meridian Ada easier.

2.4.1 Separating Specifications and Bodies

It is often useful to keep specifications and bodies of packages in separate files. Two commonly used source file extensions are:

`.ads`	package specification
`.adb`	package body

The primary advantage to keeping package specifications and bodies separate is that if only the body is changed, but the specification remains the same, then it is unnecessary to re-compile dependent units.

When the specification is kept in a separate file from the body, a potentially large amount of time is saved in terms of re-compilation and library updating if a change is made to the body that does not affect the specification.

Whenever a specification is compiled, the library entry for the specification is updated. This causes all units depending on that specification to be marked as obsolete, and the units must be re-compiled. If the specification and body are kept in the same file, but only the body is changed, then the specification continues to be updated for each compilation, making all dependent units obsolete, and requiring otherwise unnecessary re-compilations of those obsolete units.

Like ordinary package specifications and bodies, generic compilation unit specifications and bodies can appear in separate files; however, an instantiation of a generic compilation unit must occur after the compilation of the generic's body. (See Chapter 5 for more information.)

For more information about the library management system, refer to Chapter 8.

2.4.2 Reducing Program Size

One way to reduce program size significantly is to suppress automatic checks by using pragma `suppress` or the `-fs` flag to the `ada` command. Programs in which these checks have been suppressed also run much faster. Note, however, that suppressing automatic checks sometimes hampers debugging efforts because the corresponding "exception never handled" messages are then not printed (see section 3.5).

Appropriate checks to suppress include:

access_check	length_check
discriminant_check	range_check
index_check	storage_check

These checks are described in section 11.7 of the LRM.

Presently, division_check and overflow_check must be suppressed via a compiler flag, `-fN`; pragma `suppress` does not work on these two numeric checks. Refer to section 19.1 for descriptions of these flags. The on => parameter to pragma `suppress` is presently ignored; a suppressed check applies to all types within the scope of the pragma.

Another way to reduce code size is to omit the `-fL` flag, also described in section 19.1 Refer to sections 3.5 and 19.1 for descriptions of the effects of the `-fL` flag.

2.4.3 Integer Size and Portability

Programs that must be portable should not rely on the default ranges of values that can be represented by `integer` or `long_integer` objects; instead, explicit integer type definitions (ranges) should be used to declare such objects:

```
type int8 is range -128 .. 127;
  -- 8-bit integer type

type uns8 is range 0 .. 255;
  -- 8-bit unsigned type

type int16 is range -32_768 .. 32_767;
  -- 16-bit integer type

type uns16 is range 0 .. 65_535;
  -- 16-bit unsigned type

type int32 is range -2_147_483_648 .. 2_147_483_647;
  -- 32-bit integer type
```

Integer size problems usually affect ports of applications from larger systems (e.g. with 32–bit standard integers) to smaller systems (e.g. with 16–bit standard integers).

2.4.4 Catenation (&) and Storage Reclamation

When the catenation operator "&" is used to concatenate two dynamically–sized objects, the compiler must allocate temporary space to store the result. The temporaries are reclaimed on block boundaries. That is, when the statement part of a subprogram, package, task, or block is exited, the storage for any temporaries allocated by the compiler is released. This means that, if temporaries are being allocated in a loop statement, memory can be exhausted before the loop is complete. To reclaim the storage sooner, simply enclose the catenation operation in a block statement as follows:

```
with text_io;
procedure concat_example is
begin
  for i in 1..10_000 loop
    begin
      text_io.put_line("The value is " & integer'image(i));
      -- temporary storage is released here
    end;
  end loop;
end concat_example;
```

In this example, the argument to `text_io.put_line` causes temporary storage to be allocated for the result of the catenation. The begin–end block surrounding the call causes the temporary storage to be released immediately. Without the block, `storage_error` might be raised at some point when all free memory has been consumed by temporaries.

2.5 Tutorial

This section shows how to take a small programming project from start to finish using the Meridian Ada compiler and associated facilities. If you have any questions about a specific step, refer to the earlier sections in this chapter.

Note: If you are using ACE, you may prefer to use the tutorials located in the *Meridian ACE User's Guide*.

1. Key in the Ada program shown in Figure 2.1 into a file named `for_demo.ada` in your current working directory. For more information on entering a program, see section 2.2.1.

```
with ada_io;  use  ada_io;
procedure for_demo is
begin
  for i in 1..9 loop
    for j in reverse 1..i loop
      put(j);
    end loop;
    new_line;
  end loop;
end for_demo;
```

Figure 2.1 Sample Compilation Unit

2. Create a program library that references the standard library database file by running the program **newlib**. To run **newlib**, type **newlib** and press ENTER as in the following example:

   ```
   newlib
   ```

   This creates the program library file, **ada.lib**, and enters a link to **paclib\ada.lib**. For more information, see section 2.2.3.

3. Once the program library has been created with the necessary information, the sample program can be compiled. For more information, see section 2.2.4. Invoke the compiler with the **ada** command, as in this example:

   ```
   ada for_demo.ada
   ```

   If the sample program requires no corrections and recompilations, the final step is to invoke the Meridian Ada linker, **bamp**.

4. To link the components of the program together, use the **bamp** command as in this example:

   ```
   bamp for_demo
   ```

   The linker finds all the components of the program, whether visible or invisible, and binds them together to produce the executable program. For more information, see section 2.2.5.

5. The executable program file produced by the linker is given the name **for_demo.exe**. To run the executable program file, give the command:

   ```
   for_demo
   ```

   When compiled, linked, and run, this program produces the following output:

   ```
   1
   21
   321
   4321
   54321
   654321
   7654321
   87654321
   987654321
   ```

Chapter 3 Troubleshooting

This chapter provides help for some commonly encountered problems. Your first resort should be to read any *Release Notes* that accompany the compiler. These *Release Notes* sometimes contain some last minute documentation updates. Additionally, the installation chapter located in the *Getting Started* manual, contains installation specific troubleshooting.

3.1 Compiler Won't Run

The compiler won't run (e.g. it simply returns to the system prompt without doing anything).

3.1.1 Nothing Happens

Possible solutions:

- Make sure that you have typed an extension (`.ada`) with the file name to be compiled.
- Make sure that `path` is set correctly (see the installation instructions); type the command `path` to find out. Rebooting the system may cause whatever automatic startup procedures that are already in place to set the `path` correctly.
- Make sure that there are no resident programs running (print spooler programs, window managers, etc.).
- Make sure that a "subtle" machine crash has not occurred previously. This should not happen, but under some rare circumstances, the compiler may handle program errors incorrectly. If this happens, the system is left in an undesirable state. Such problems should be reported in writing to Meridian Software Systems with a complete description of the machine and operating system environment and the circumstances under which the problem occurred. To correct this problem, reboot the system.
- Make sure that `files=20` in the `config.sys` file. See the installation chapter for more information.

3.1.2 Cannot Exec

You receive the error message:

```
cannot exec to path\ada1.exe
```

This problem usually occurs when, for some reason, there is not enough memory available to execute the compiler.

Possible solutions are:

- Make sure that there are no resident programs running (print spooler programs, window managers, etc.). This may include network drivers and unusually large device drivers that consume portions of base memory (the main 640K area).

- Try rebooting. This may release memory which some previously run editor program or other utility continues to hold.
- Ensure that **ada1.exe** actually exists in the *path* specified in the error message. If it doesn't, it may mean that the compiler was not installed correctly, or that files have been moved around or renamed.
- Check your **autoexec.bat** file to see what programs are invoked whenever your system is booted. You should eliminate any invocations of terminate–and–stay–resident programs (programs that stay around in memory, executing in the background). This includes mouse programs, "hot key" programs, and other utilities that execute in the background. Once you have modified your **autoexec.bat** file, you must reboot.
- Check your **config.sys** file to see what drivers are installed whenever your system is booted. If there are many lines beginning with **device**, then you must eliminate some or all of them. The **ansi.sys** driver is small enough that it can usually remain. Once you have modified your **config.sys** file, you must reboot.
- If you have more than 1MB of memory installed in your system and you have installed the Extended Mode compiler, try using that compiler instead. Note: This applies only to extended memory, and not expanded (also called LIM) memory. See Chapter 11 for a discussion of extended mode versus expanded mode.

3.2 Compiler Out of Memory

If the compiler runs out of memory for symbol table space (this is often plain to see if the **ada -fv** option is used), then compilation naturally halts. This problem might be solved in any of several ways:

- Ensure that no other resident programs are present. If other applications are in memory (e.g. print spoolers, window management systems, spread sheet programs,etc.), they should be terminated. The system may have to be rebooted in order to clear some applications from memory.
- Use the batch version of the compiler, **ada2**, described in section 19.2.
- Check your **autoexec.bat** file to see what programs are invoked whenever your system is booted. You should eliminate any invocations of terminate–and–stay–resident programs (programs that stay around in memory, executing in the background). This includes mouse programs, "hot key" programs, and other utilities that execute in the background. Once you have modified your **autoexec.bat** file, you must reboot.
- Check your **config.sys** file to see what drivers are installed whenever your system is booted. If there are many lines beginning with **device**, then you must eliminate some or all of them. The **ansi.sys** driver is small enough that it may usually remain. Once you have modified your **config.sys** file, you must reboot.
- If you have more than 1MB of memory installed in your system and you have installed the Extended Mode compiler, try using that compiler instead. Note: This applies only to extended memory, and not expanded (also called LIM) memory. See Chapter 11 for a discussion of the difference.
- Reduce the number of symbols that appear in package interfaces. The more symbols that appear in the body of a package rather than in the interface, the better.

When a package is with–ed, all the symbols that appear in its interface consume long–term symbol table storage that is not released until the end of the compilation unit.

- Localize as many symbols as possible The more symbols that appear inside subprograms and declare blocks, particularly in package bodies, the better. Symbol table space consumed by a local symbol is released at the end of the scope in which the local symbol appears. This allows the same symbol table space to be reused over and over.
- Split the compilation unit into several smaller compilation units.
- Split the program into several smaller co–operating programs. This may require use of the DOS **exec** function. The *DOS Environment Library* provides this capability.

Note: If **bamp** similarly runs out of memory, there is a "virtual mode" option for the linker that may help to solve this problem.

3.3 Cannot Find Kernel

You receive the following error message:

```
OS/x86: cannot find operating system kernel
```

This error occurs when the Extended Mode MeridianAda compiler was not installed properly. Consult the installation chapter in the *Getting Started* manual for more information.

3.4 Library Problems

3.4.1 Main Program Not in Library

You receive the following error message:

```
main program xxx is not in the library
```

Possible solutions are:

- Type the **lslib** command to find out what exactly is in the library.
- Make sure that you actually compiled the program in question.
- Make sure that you did not type an extension (e.g. **simple.ada**) to the **bamp** command. Be careful not to confuse the file name (**simple.ada**) with the compilation unit name (**simple**).

3.4.2 Missing Library Unit

You receive the following error message:

```
missing library unit x
```

This message appears during compilation when a unit is with–ed that cannot be found in the program library or in any of the libraries to which direct links have been established (see Chapter 8).

Possible solutions are:

- Make sure that you have used the **newlib** command in the current directory. A file named **ada.lib** should be present in the directory if this has been done.
- Make sure that you have established all necessary links between libraries. Use the command **lslib -k** to determine exactly which links are present in the program library. If the necessary link is missing, it should be added with **lnlib**.

- Make sure that you have typed the name of the unit correctly in the with clause.
- Make sure that the unit in question is actually present in the program library or in any of the libraries to which direct links have been established. First use the **lslib -k** command on the program library and then use the **lslib** command on each library shown in the list of links.
- Make sure that the unit in question was actually compiled.

3.4.3 Wrong Version in Library

You receive the following error message:

wrong version in library

This error message appears when mismatched library version number stamps are detected, as when an obsolete library (a library built with a previous release of the compiler) is linked with a newer library.

Possible solutions are:

- Make sure that you are using the current version of the **newlib** program.
- Make sure that **path** is set correctly and contains the new Meridian Ada **bin** directory and not the old Meridian Ada **bin** directory. Type the command **path** to find out.
- Make sure that no link entries to obsolete libraries are present in your program library. Find out what links are present with the **-k** option to **lslib**; get rid of the obsolete links with the **-r** option to **lnlib**.

3.5 Exception Never Handled

When an exception propagates out of the main subprogram, an error is printed at run–time:

Exception never handled: *exception–name*
problem–description **in** *file–name*, **line** *line–number*

The *problem–description* information provides valuable information about how and where an exception was raised, simplifying program debugging efforts. Note that the *problem–description* does not appear unless the compiler option **-fL** was given (see section 19.1).

The *file–name* and *line–number* information is printed only for exceptions raised in user–written programs. The file and line information is not printed for exceptions raised within pre–defined packages such as **text_io** (this is likely to reduce possible confusion about the origin of an exception).

Only programs that use **text_io** can produce "Exception never handled" messages for exceptions that propagate out of the main program. This means, for example, that programs using package **tty** from the *DOS Environment Library* for I/O instead of the standard **text_io** package cannot produce the run–time error message. Exceptions may still be handled in the usual way: only the error message is suppressed. If any compilation unit anywhere in a program uses **text_io** then the "Exception never-handled" message can be produced. Programs that use the supplementary packages **ada_io**, **iio**, or **fio** are actually using **text_io** indirectly, thus such programs can produce the error message.

3.5.1 Automatic Checks

For exceptions generated by automatic checks, *problem–descriptions* include:

- **Assignment would change discriminant**

- **Constraint clash for access type**
- **Floating point constraint clash**
- **Floating point divide by zero**
- **Floating point value out of range**
- **Illegal record variant**
- **Length clash in multi-dimensional array**
- **Length or discriminant clash** *actual-value /= required-value*
- **Reference through null access value**
- **Value** *actual-value* **out of range** *first..last*

For value out of range errors, the first and last values of the range are presently printed as integers, even for enumerations and fixed-point values.

"Exception never handled" messages produced by automatic checks may be suppressed by appropriate use of pragma **suppress** (see section 2.4.2). Note that suppressing automatic checks does not affect any explicit raise statements (i.e. an "Exception never handled" pops up if indeed the exception raised is never handled).

Some pre-defined exception names used by automatic checks are:

```
constraint_error        storage_error
numeric_error           tasking_error
program_error
```

The meanings of these exceptions are defined in the LRM.

3.5.2 Storage_Error

The **storage_error** exception may be raised, among other reasons, when a local object (an object defined inside a procedure, function, or task) is too large. "Too large" here means that the local object cannot be accommodated in the available stack space; however, increasing the amount of stack space may not be the entire solution. The amount of stack space available at a particular moment varies according to the amount of stack space consumed by each subprogram activation. Space consumed by an activation includes the space for the parameters and local objects. For example, a local array is going to consume space on the stack when the subprogram containing the array is called. That space is reclaimed when the subprogram returns, but if the subprogram is called recursively, or many levels of subprogram calls are involved, then the additional amount of stack space consumed by each call must be considered. Chapter 13 has some additional information about stack space limitations and how to work around them.

3.5.3 Other Exceptions

A number of other exceptions used by the I/O run-time packages are defined in package **io_exceptions**, described in the LRM, section 14.4. I/O exceptions include:

```
data_error              mode_error
device_error            name_error
end_error               status_error
layout_error            use_error
```

Explanations of the implementation-dependent circumstances under which these I/O exceptions are raised are given in this document in Chapter 4.

Other exceptions can be programmer defined.

Chapter 4 Input–Output

This chapter provides information on implementation–dependent aspects of Meridian Ada input–output (I/O) on PC–DOS. Support packages and generic packages that are affected by implementation–dependencies in the I/O are `text_io`, `direct_io`, and `sequential_io`.

The low–level run–time I/O support packages use the PC–DOS file system services. Although this chapter contains summaries of how this run–time structure affects input–output operations in Ada programs, more specific information about the low–level mechanisms is provided in the PC–DOS technical reference documents.

In the PC–DOS configuration of Meridian Ada, external files are manifested as sequential files associated with devices (e.g. `CON`, `PRN`, `COM1`) or as sequential and direct files resident on disk devices.

At a low level, there is little need to treat PC–DOS files differently based on their contents. For example, files containing human–readable text are treated no differently from files containing object code libraries. All disk files are treated simply as arrays of bytes (in `direct_io`) or as byte streams (in `sequential_io` and `text_io`).

4.1 Standard_Input and Standard_Output

The files returned by the functions `standard_input` and `standard_output` are associated with the PC–DOS standard input and standard output streams. The standard output stream by default references your terminal output device (typically a CRT terminal device named `CON`). The standard input stream, by default, references your keyboard device (also named `CON`).

On PC–DOS, the standard input and output streams may be re–directed at program invocation time, as described in the PC–DOS technical documentation. To summarize briefly, re–direction of the standard I/O streams is achieved on the command line by using these special command forms:

$program_1$ | $program_2$	to pipe output to another program
program < file	to re–direct input from a file
program > file	to re–direct output to a file
program >> file	to re–direct output and append to the end of an existing file

Note that it is possible to re–direct output to a device as well; re–directing output to the file named `PRN` sends output to the printer device.

4.2 Terminal I/O

Terminal input and output is *line buffered*. That is, input is not seen by a program until a new–line is entered, and output is not visible on the screen until either a new–line is printed or a terminal input operation is performed (so that same–line prompting works). This means that reading a character–at–a–time from the terminal may not work as expected, since a new–line must be entered by you for the program to see the input. "Raw" character input from the terminal can be done via the *DOS Environment Library* using `tty.get` in order to get immediate character–at–a–time response.

Greater control over output buffering can be achieved using the Meridian Ada Utility package `spio`. For more information see section 18.6.

4.3 Opening and Closing Files

This section discusses implementation–dependencies related to opening files with **create** and **open** calls, and also closing files with **close**.

4.3.1 File Name Format

File name strings supplied to the **create** and **open** procedures have the forms permitted on PC–DOS. Although these forms are documented in the PC–DOS technical references, a summary is given here, followed by examples. For more information see your *DOS User's Guide*.

Briefly, a file name has the format:

d:path\name.ext

as in

```
c:\ada\test\simple.ada
```

where:

d: The *d* (disk) portion of a file name is a letter in the range 'A'..'Z' followed by a colon (':'). The disk may be omitted, in which case the default disk referenced is the currently selected disk, whose letter is usually displayed at the command line prompt.

path \ The *path* portion of a file name is a directory path specification terminated by a backslash ('\'). An explanation of the hierarchical organization of PC–DOS directories is beyond the scope of this document. Briefly, a directory path specification is a sequence of names (possibly with extensions (*.ext*)) separated by backslash characters. A directory path specification indicates a hierarchically–ordered list of directories that must be traversed to find the file whose name follows the path. The path may be omitted, in which case the current working directory is used.

name The *name* portion of a file name is one to eight characters long, optionally followed by a period (".") and an extension. Characters legal in *names* are:

- The letters 'A' through 'Z'.
- The decimal digits '0' through '9'.
- The punctuation characters "$&#@!%()–{}".
- The underscore character, '_'.
- The single quote character, ' '.
- The accent grave character, ' ` '.

The lower case letters 'a' through 'z' are treated as their upper case equivalents. Other characters terminate a file name.

.ext The optional *ext* portion of a file name is one to three characters long, preceded by a period ('.'). Restrictions on characters in the *name* portion of a file name apply also to the extension. The extension may be omitted from a file name. If the extension is omitted, the period may be omitted as well. There is no default extension.

Note the placement of colon (':') and period ('.') separators.

Inside an Ada program, the extension is really considered as just another part of the *name*, although external files are categorized by extensions. Some examples of extensions and the categories represented are:

.ada, .ads, .adb, .sub
Meridian Ada source files

.asm	assembly language source files
.atr	interface description ("attribute") files
.bat	batch command files
.exe	Real Mode executable load modules
.exp	Extended Mode executable load modules
.gnn	generic description files
.obj	object files
.sep	subunit ("separate") environment description files

Some extensions have special meanings to the operating system (e.g. **.exe, .bat**), while others have special meanings only to specific application programs (e.g. **.ada, .atr,** ***.gnn*** for Meridian Ada). No extension is assumed or required by Meridian Ada I/O.

Default File Name Components

Select a disk prior to running a program by specifying its letter followed by a colon, as in this example:

```
a:
```

A disk may also be selected as the default disk from a program by issuing the appropriate PC–DOS system call.

The PC–DOS command **cd** is used to select the current directory, as in this example:

```
cd \ada\test
```

The current directory may also be selected from a program by issuing the appropriate PC–DOS system call.

Note that a separate current directory is maintained for each disk, and the current directory changes when the default disk changes.

Ada system interface functions to call PC–DOS operating system services, such as those that change the default disk and directory, are available in the *DOS Environment Library*. A program need not change its current directory or default disk if the necessary selections are made before the program is run or if complete file name specifications are given.

Refer to the PC–DOS technical reference manuals for specific information on operating system commands and service functions.

Special File Names

Note that there are special file names that refer to non–disk devices:

CON	the console
AUX	serial port 1
COM1	same as AUX
COM2	serial port 2
LPT1	parallel port 1
PRN	same as LPT1
LPT2	parallel port 2
LPT3	parallel port 3
NUL	null (bit–bucket) device

Non–sequential (direct) input–output cannot be applied to any of these devices. Refer to thePC–DOS technical reference manuals for additional information about the characteristics of these devices. Note that there may be site–specific reserved device names.

To open one of the special files for output, use **create** with mode **out_file** (the default mode) as in the following example:

```
with text_io;  use text_io;
procedure printer_demo is
  f: file_type;
  printer: constant string := "PRN";
begin
  create(file => f, name => printer);
      -- Open the printer device for output.
      -- Note: assumes that the device is pre-initialized.
  put_line(file => f, item => "This is a test.");
  new_page(file => f);
      -- Page eject may or may not be
      -- appropriate for any given printer.
  close(file => f);
end printer_demo;
```

This example prints on the printer device the line "**This is a test.**" followed by a page eject, an ASCII FF (form feed) character. Note that although most printers recognize the standard ASCII FF character as the page eject, some printers may not.

To open one of the special files for input, use open with mode **in_file** as in this example:

```
with text_io;  use text_io;
procedure read_com is
  f:    file_type;
  com:  constant string := "COM1";
  line: string(1..256);
  last: natural;
begin
  open(file => f, name => com, mode => in_file);
      -- Open the communications port for input.
      -- Note: assumes that device is pre-initialized
      -- with respect to rate, stop bits, etc.
  while not end_of_file(f) loop
    get_line(file => f, item => line, last => last);
    put_line(line(line'first..last));
  end loop;
  close(file => f);
end read_com;
```

This example reads lines from the communications port (under certain idealized conditions) and prints the lines on the standard output stream. Note that this example may not work under all circumstances related to the communications port.

File Name Examples

This section provides some examples of string literals that specify PC–DOS file names.

c:\bin\f.ini	This is an example of a fully specified path, including drive specifier. It refers to a specific file in a specific directory on a specific disk.
template	A file name specified in this form, without directory or drive specifier, refers to a file in the current working directory on the currently selected disk.

`c:x` Because DOS maintains a separate current working directory for each disk, a file name specified in this form refers to a file in the current working directory on a specific disk.

4.3.2 File Forms

File form parameters to **create** and **open** must be empty strings (""). If a file form is not empty, the exception **use_error** is raised. Note that the default values of file form parameters to **create** and **open** are in fact empty strings. An example follows.

```
with text_io;  use text_io;
procedure file_form_demo is
  f: file_type;
begin
  open(form => "",  -- Must be empty or omitted altogether.
       file => f,
       name => "data",
       mode => in_file);

  close(file => f);
end file_form_demo;
```

4.3.3 Limitations on Opening Files

Sharing External Files

You can have several file objects opened with mode in_file on a single external file.

You cannot have several file objects opened with mode out_file or inout_file on a single external file.

Number of Open Files

PC–DOS by default limits the number of open files to 8, including five pre–defined device file associations (two of which are used as **standard_input** and **standard_output** by Meridian Ada programs). Note that this default can be changed at any given site by using the **files** command in the PC–DOS configuration file, **config.sys**. Remember, no process can have more than 20 files open at one time (including the five pre–defined device file associations), and no more than 99 files can be open at one time throughout the system. Refer to the PC–DOS technical reference documents for more specific information regarding these limitations.

Buffering Considerations

The effect of opening a device as a file is to cause buffered input or output operations, as with disk files. I/O on a terminal device is line–buffered. Refer to section 4.2 for additional information about the way terminal I/O works.

4.3.4 Leaving Files Open

When a disk file remains open after completion of the main program, it disappears if it was created with an empty name (""). Otherwise, output file buffers are properly flushed and the files closed. Note that a system crash leaves output files in an indeterminate state.

4.3.5 Creating Files

Files created with **create** are initially empty, zero–length files. Note that files cannot be created with **open**. Other than described in the previous sections, there are no special considerations for opening files.

All files are created with normal attributes:

- read/write (as opposed to read–only)
- visible (as opposed to hidden)
- non–directory (as opposed to directory)

These defaults are acceptable for almost all applications except those few that are system–oriented. Note that the *DOS Environment Library* provides facilities for Meridian Ada programs to get and set specific file attributes.

4.3.6 Closing Files

There are no special considerations for closing files, except that it is a good idea always to close a file upon completing I/O operations on the file. Otherwise, it is possible for a program to run into the limitations on the number of open files. Also, it increases the portability of a program; not all Ada implementations handle unclosed files upon program termination as gracefully as the Meridian Ada run–time support does.

4.4 Terminators

A `text_io` file terminator is the physical end of a file as defined by the PC–DOS operating system. Although the Control–Z (ASCII SUB) character is used as a text file terminator, it is discarded on input (i.e. it is not seen by an Ada program). A Control–Z is never written to an output file.

A line terminator in a text file is a carriage return (Control–M, ASCII CR) character followed by a line feed (Control–J (^J), ASCII LF) character. The end of line condition is also set when a file terminator or page terminator is encountered.

A page terminator in a text file is a form–feed (Control–L (^L), ASCII FF) character. The end of page condition is also set when a file terminator is encountered. To make the interactive behavior of `end_of_page` sane, there is no wait for a page terminator on terminal input when `end_of_page` is called, but when a page terminator is encountered, the necessary processing takes place.

4.5 Other File Operations

This section discusses implementation–dependent aspects of other file operations. Exceptions raised by file operations are discussed in section 4.7.

4.5.1 Reset

Permanent Files

Any type of permanent file can be reset to any mode. Whenever a permanent text file is reset to `out_mode`, all existing data in the file is destroyed.

Temporary Files

The following table indicates under what circumstances the mode of a temporary file may be reset. The letters 'S', 'D', and 'T' have these meanings:

S permitted with sequential files
D permitted with direct files
T permitted with text files

The absence of any letter means that the new reset mode is not permitted with that file type. The word "never" means that the new reset mode is not permitted with any file types.

original open mode	new reset mode		
	in_file	out_file	inout_file
in_file	SDT	never	never
out_file	SDT	SD	D
inout_file	D	D	D

To summarize the table, reset can not be used to change the mode of a temporary file of mode in_file. A sequential file of mode out_file can be changed only to an in_file. A direct file of mode out_file or inout_file can be changed to any mode.

4.5.2 Name

Using the function `name` with the `standard_input` and `standard_output` files returns `"STANDARD_INPUT"` and `"STANDARD_OUTPUT"`.

4.5.3 Form

Function `form` always returns an empty string ("").

4.6 Low Level I/O

The pre–defined Ada package `low_level_io` is not implemented. There are ways to make `low-level` access to system supported devices through facilities in the *DOS Environment Library* such as `absolute_disk`, `file_io`, `port`, `tty`, and `video`.

4.7 Exceptions

There are no special I/O exceptions for PC–DOS.

Here are the implementation–dependent circumstances under which specific exceptions are raised:

`data_error` The `data_error` exception is raised when there is insufficient data in the file to fill up an element of the specified type.

`device_error` `Device_error` is raised for:

- *insufficient memory:* The operating system cannot find enough free memory to perform an I/O operation.
- *invalid data*: Invalid information is read from the device on which the external file resides (as determined by PC–DOS).
- *internal DOS error*: Any number of internal errors are discovered by the operating system.
- *illegal deletion*: An attempt is made to remove the current directory.

name_error The **name_error** is raised for:

- *no such file*: The file to be opened does not exist.
- *missing file*: A file (other than a temporary file) that must exist before being opened cannot be found with the specified name or with the specified path.
- *illegal name*: The name of the file to be opened is not acceptable to PC–DOS.
- *illegal name*: The file name given to **open** or **create** is too long (more than the maximum number of characters accepted by DOS–usually 64–or more than 255 characters, whichever is shorter).

use_error The **use_error** exception is raised for:

- *missing temporary file*: A temporary file that must exist before being opened or deleted cannot be found with the internally–supplied name or with the specified path.
- *access denied*: The specified file cannot be opened or access to the file is denied for some reason.
- *file system full*: The file system cannot accommodate any new files.
- *illegal form*: A non–empty form string is given.
- *illegal reset mode*: An open file is reset to an illegal new mode (see section 4.5.1).

I/O exceptions are also raised under other circumstances defined by the LRM.

4.8 I/O of Unusual Types

4.8.1 Unconstrained Objects

Packages **direct_io** and **sequential_io** must only be instantiated on fixed–size objects. I/O of unconstrained objects (e.g. strings and unconstrained discriminant records) is not directly supported. For example:

```
-- The Wrong Way:
with sequential_io;
package wrong_string_io is new sequential_io(string);
    -- Error: cannot instantiate with unconstrained type.

-- One Correct Way:
with sequential_io;
package the_right_way is
  subtype typical_string is string(1..80);
  package right_string_io is new sequential_io(typical_string);
end the_right_way;

-- Another Correct Way:
with text_io;  use text_io;
-- See below.
```

Note that although **direct_io** and **sequential_io** for the unconstrained type string, as in the above example, is not supported as such, **text_io** provides all the necessary facilities for input and output of strings.

As with unconstrained array types (e.g. `string`), `direct_io` and `sequential_io` cannot be instantiated for an unconstrained discriminant record type. Note that a discriminant record with default expressions for the discriminants is still considered to be unconstrained. Although an instantiation of `direct_io` and `sequential_io` on such a record type compiles, a `use_error` exception is raised at run–time. Also note that using a constrained subtype of a discriminant record or using a record altogether without discriminants is acceptable. An example follows.

```
with direct_io;
package io_example is
  type selector is (iv, cv, bv);
  type vrec(vselect: selector) is
    record
      case vselect is
        when iv => i: integer;
        when cv => c: character;
        when bv => b: Boolean;
      end case;
    end record;
  subtype i_vrec is vrec(vselect => iv);
  subtype c_vrec is vrec(vselect => cv);
  subtype b_vrec is vrec(vselect => bv);
  package i_vrec_io is new direct_io(i_vrec);
    --This is okay, because i_vrec is constrained.
end io_example;
```

4.8.2 Access Objects

Input–output for access types, although permitted, is normally a bad idea. An access object previously written to a file may be retrieved, but the access object should only be used if the access object is still valid. An access object is valid only until completion of the program that allocated the object referenced by the access object, or until the object referenced is de–allocated, whichever occurs first. An example follows.

```
with direct_io;
with iio;
procedure access_io is
  type ip is access integer;
  package ip_io is new direct_io(ip);
  use ip_io;
  i: ip;
  f: file_type;
begin
  create(file => f, name => "icky");
  i := new integer'(2);
  write(file => f, item => i);
  set_index(file => f, to => 1);
  read(file => f, item => i);
  iio.put(i.all);
  close(file => f);
end access_io;
```

This example assigns a value to an integer access object, writes it to a file, retrieves it, and writes the value of the object to which it points. Refer to section 17.1 for information about package `iio`.

4.9 Implementation–Defined Types

In the specification of `direct_io`, the type `count` is implementation–defined as:

```
type count is range 0 .. long_integer'last - 1;
```

In the specification of `text_io`, the types `count` and `field` are implementation–defined as:

```
type count is range 0 .. integer'last - 1;
subtype field is integer range 0  ..  integer'last;
```

Chapter 5 Generics

5.1 Implementation

In the Meridian Ada compiler, generics are implemented so that they expand fully each time they are instantiated (this is sometimes called a "macro" implementation of generics). While this implementation of generics tends to be very time–efficient, it is not necessarily space–efficient. This means that a generic should be written so that as much common code as possible is placed outside the generic unit. Also, a generic should be instantiated infrequently to reduce the amount of code generated. Chapter 17 demonstrates some of this wisdom as applied to input–output generics.

5.2 Restrictions

The specification and body of a generic compilation unit can appear in different compilation files. This also applies to any generic body subunits; however, if this is the case, then any instantiations of such a generic must be compiled after the compilation of the generic's body and any associated subunits. If this order of compilation is violated, then an error is generated when the Ada program is linked with **bamp**. The error indicates that the compilation unit containing the instantiation is obsolete and must be recompiled.

The way generics are implemented in Meridian Ada affects certain subtle compilation order requirements of the Ada language definition. The compilation unit instantiating a generic depends on the generic's body and any associated subunits, as well as the generic's specification. This dependency is necessary because the validity of the instantiation is determined, in part, by the contents of the generic body.

Chapter 6 Tasking

This chapter discusses the Meridian Ada implementation of tasking.

There are two distinct task scheduling methods for Meridian Ada, preemptive and non–preemptive. Normally, the task scheduler is non–preemptive (e.g. time slicing is not supported). If the **-I** option is specified to **bamp**, a different preemptive scheduler is used. Preemptive scheduling allows quicker servicing of interrupts on delay timeouts, but has a slight negative effect on program performance in general.

The priority of a task is fixed at compile–time with pragma **priority**. The range of priority values is given in the integer subtype **system.priority** (see Appendix F). The default priority is undefined, as in the LRM. It so happens that in Meridian Ada tasking, a task with an undefined priority runs at a lower priority (a lesser degree of urgency) than any tasks with defined priorities.

6.1 Non–Preemptive Tasking

The non–preemptive tasking run–time uses a round–robin prioritized scheduling algorithm that switches tasks only at activations, entry calls, completions, and wait conditions.

In non–preemptive mode, tasks in infinite loops that encounter no tasking constructs inhibit task switching altogether. One way to prevent this is to put a **delay** statement into the loop:

```
delay 0.0;
```

When a negative delay or a delay of 0.0 is encountered in a task, that task is rescheduled. Any other tasks of equal or greater priority will then run first.

An example:

```
with text_io;  use text_io;
procedure reschedule_demo is
  task type customer;
  task type teller is
    entry deposit;
    entry withdraw;
  end teller;

  harried_teller: teller;
  hurried_customers: array(1..2) of customer;
```

```
task body teller is
  customers_remaining: boolean := false;
begin
  put_line("teller activated");
  loop
    select
      accept deposit do
        put_line("deposit");
      end deposit;
    or
      accept withdraw do
        put_line("withdraw");
      end withdraw;
    else
      null;
    end select;
    customers_remaining := false;
    for i in hurried_customers'range loop
      customers_remaining :=
         customers_remaining or else
         not hurried_customers(i)'terminated;
    end loop;
    exit when not customers_remaining;
  end loop;
  put_line("teller closes window");
end teller;
task body customer is
begin
  put_line("customer activated");
  harried_teller.deposit;
  delay 0.0;  -- reschedule
  harried_teller.withdraw;
  delay 0.0;  -- reschedule
  harried_teller.withdraw;
  put_line("customer closes account");
end customer;
begin
  -- Harried_teller & hurried_customer tasks activated here.
  null;
end reschedule_demo;
```

6.2 Preemptive Tasking

The preemptive tasking run–time supports the following run–time features:

1. More immediate preemption among tasks of different priorities due to interrupts.
2. Time slicing among tasks of the same priorities.

The sections which follow describe preemption and time slicing, the interface to these features, some programming hints for using task preemption and time slicing, and some limitations associated with the current implementation.

6.2.1 What is Task Preemption?

Task preemption is the ability to switch from running a lower priority task to a higher priority task as soon as the higher priority task becomes "ready to run". This task switch might occur at any point in the execution of the lower priority task. The higher priority task might become "ready to run" due to the expiration of a delay statement, or due to the occurrence of an interrupt for which the task has defined an interrupt entry.

For example, suppose the higher priority task is waiting for a delay statement to expire. When the last tick of the clock occurs that causes the delay to time out, the lower priority task might be in the middle of executing the following Ada assignment statement.

```
count := count + 1;
```

In fact, the machine instructions generated for this statement might have just loaded the value of `count` into a temporary register and incremented the register by one, but NOT stored the new value back into `count` when the delay expires. Given this situation, task preemption would cause the following to happen, all before the value of `count` is updated.

1. The execution of the lower priority task would be suspended.
2. The task's execution context (the values in its registers and other task–specific run–time information) would be saved.
3. The tasking run–time would restore the execution context of the higher priority task.
4. The higher priority task would resume its execution and continue until it had to wait again (maybe for another delay statement or an entry call).
5. Eventually, the lower priority task would become the next available task to run.
6. The lower priority task's context would be restored and its execution would be allowed to continue.

Only now would the updated value of `count`, which was still stored in a temporary machine register, finally be stored into the `count` variable itself.

6.2.2 What is Time Slicing?

Time slicing is the ability to switch between two tasks of equal priority when both tasks are "ready to run". The previous section focused only on the situation where a lower priority task was preempted by a higher priority task. This is the minimum degree of preemption dictated by Section 9.8 (Priorities) of the LRM. However, the Ada language does not define what should happen when two or more tasks of equal priority are ready to run: which one should run first and for how long? The question of which to run first is typically not important but how long it should run is. The LRM does have something to say about how long a task can keep other tasks of equal priority from running, but only at what are generally called synchronization points. A synchronization point is one of the following:

- The end of a task's activation
- A point where it causes the activation of another task
- An entry call
- The start or the end of an accept statement
- A select statement
- A delay statement
- An exception handler
- An abort statement

How long a task can run without reaching these points is undefined by the LRM.

For example, the current task might be executing the following loop.

```
loop
  ready := true;
end loop;
```

Such a loop, of course never exits and, in particular, it never reaches a synchronization point. Given the rules of Ada, even if such a task is preempted and aborted by a higher priority task, the task need not be terminated and need not ever give up control to any task of equal or lower priority. Although this is an extreme example of how one task can lock out other tasks from executing, it illustrates a situation which is not uncommon in real–life tasking programs.

To remedy this situation, Meridian Ada preemptive tasking run–time has the capability of performing time slicing. Time slicing gives you limited control to add a degree of fairness to the task scheduling algorithm so that tasks of equal priority are not forever locked out of running. With time slicing you can dictate the maximum amount of time that a given task can execute without giving up control of the processor to other tasks of equal priority.

Note that time slicing only affects the scheduling of tasks of equal priority. The LRM defines when tasks of higher or lower priority may run. In particular, in the above example with the loop statement, even if time slicing is turned on, no task of lower priority will ever get a chance to run.

6.2.3 Using Task Preemption

For an Ada program to use preemption or time slicing, the program must be linked with the **bamp -I** option. This option causes the preemptive tasking run–time to be linked into the program. Normally, without this option, a tasking run–time with more limited preemptive capabilities is linked in which performs task context switches only at synchronization points. This default tasking run–time can be used to generate a program with more predictable (repeatable) behavior and slightly less overhead. The default run–time is also compatible, in behavior, with previous releases of Meridian Ada.

The **bamp -I** option is only useful if the Ada program actually has tasks in it.

6.2.4 Package Task_Control

When a program linked with **bamp -I** first starts to run, task preemption is on, but time slicing is off. The following sections describe how these features can be controlled by you.

The following package is part of the Meridian Ada standard distribution library:

```
package task_control is

  procedure pre_emption_off;
  procedure pre_emption_on;

  procedure set_time_slice(quota: duration);

end task_control;
```

To improve portability, this package can be used by any Meridian Ada program, regardless of whether it was linked with the **bamp -I** option and regardless of whether it uses tasking. The use of the **task_control** package only affects a program that was linked with the **bamp -I** option and that does use tasking.

6.2.5 Controlling Preemption at Run–time

Recall the task preemption scenario above, where a lower priority task was in the middle of executing the following Ada statement when it was preempted by a higher priority task.

```
count := count + 1;
```

Note that, if `count` was a global variable, it is possible that its value was examined by the higher priority task while the lower priority task was suspended. If so, the value of `count` would have been the value it had before the assignment statement was executed. Furthermore, it is possible that the higher priority task modified `count` while the lower priority task was suspended. If this was the case, then the completion of the assignment statement in the lower priority task would destroy the value of `count` set by the higher priority task. For example, if `count` was 4 before the assignment statement and the higher priority task set `count` to 10, then at the completion of the assignment statement in the lower priority task `count` would be 5 and not 11. This later case is an example of a program which is formally erroneous as defined by Section 9.11 (Shared Variables) of the LRM which makes the behavior of the program unpredictable. Herein lies one of the major difficulties in using the task preemption capability correctly; you must diligently protect the use of shared variables (especially those that might be hidden inside some external package).

There are several ways to solve this problem, each with its own costs and benefits.

One way is to hide a shared resource (such as a global variable) inside the body of a package and provide only a procedural interface for accessing the resource. Then have each procedure within the package call on what is sometimes referred to as a guard task before accessing the resource to get permission to access the resource and call the guard task again after accessing the resource to release the resource for others to use. When properly implemented, this scheme should have the effect of allowing only one task at a time to execute the code within the package's procedures. This scheme is considered to be the standard, approved, way of controlling access to a shared resource within an Ada program.

Another way of controlling access to shared resources is to call the procedures in the `task_control` package to temporarily disable task preemption. While preemption is disabled a task may access any number of shared resources without having to worry about another task interfering. Although this is a rather gross way of controlling resource access (in that it can affect the execution of all tasks in a program), it requires very little overhead in terms of both space and time.

Preemption is disabled simply by calling procedure `pre_emption_off` and reenabled by calling procedure `pre_emption_on`. These procedures must be properly paired to ensure that preemption is reenabled at the right time. If not, then preemption might be reenabled too soon, or it might be disabled for the remaining life of the program. Calls to these procedures may be nested as illustrated in the following code fragment.

```
begin
   -- Preemption is assumed to be enabled here.
  task_control.pre_emption_off; -- Preemption is now disabled.
  task_control.pre_emption_off; -- Preemption is still disabled.
  task_control.pre_emption_on;  -- Preemption is still disabled.
  task_control.pre_emption_on;  -- Preemption is now reenabled.
end;
```

In general, preemption is disabled whenever the number of calls to `pre_emption_off` is greater than the number of calls to `pre_emption_on`, although there are two exceptions to this rule. First, calling `pre_emption_on` when preemption is already enabled does not affect the standing count (that is, it's a no-op). Second, if an exception is raised, then the standing count is reset back to the value it had when the block that ends up handling the exception was first entered.

Note that disabling preemption does not prevent a task from voluntarily giving up control to another task as might occur if it executes a delay statement. For this reason, it is almost certainly a mistake for code to perform any tasking operations while it has preemption disabled.

6.2.6 Controlling Time Slicing at Run-time

The `task_control.set_time_slice` procedure is used to turn time slicing on and off and to specify the length of a time slice. The length of a time slice is often referred to as the execution time quota which is

why the only parameter to this procedure is called *quota*. Like all duration values, *quota*, is given in terms of seconds. If `set_time_slice` is called with a *quota* that is less than or equal to zero (0.0), then time slicing is turned off; otherwise it is turned on. When enabled, time slicing applies to all tasks of all priorities, but remember, it only has an effect when two or more tasks of equal priority are active.

The amount of time a task gets to execute before its quota runs out, may actually be less than or greater than the value given to `set_time_slice` for various reasons. Some of these reasons are: 1) operating system delays, 2) differences introduced when converting the quota from a duration quantity to the equivalent number of system clock ticks (whose duration is defined by `system.tick`), and, 3) the disabling of task preemption. In the later case, if a task's quota expires while preemption is turned off, the current task will be suspended as soon as preemption is reenabled. In most cases the difference between the specified quota and the actual quota will be a small percentage of the specified value.

Note that the smaller the quota, the greater the amount of overhead added to the program due to the increased frequency of task context switching.

6.2.7 Limitations and Warnings

The Meridian Ada run–time manages many resources which can be shared among many tasks. Some of these include, I/O resources (files and devices), memory resources (allocating access variables), and CPU resources (task scheduling). In order to maintain the integrity of these resources, the run–time code which manages these resources is protected from being preempted when preemption is enabled with the `bamp -I` option. In particular, since the I/O run–time is protected from being preempted, input operations via `text_io`'s standard input will suspend all other tasks' execution until the operation is completed.

When running a preemptive tasking program under the debugger, preemption is always disabled while at the debugger's command level (that is, while the program is suspended at a breakpoint). When the program is running, however, preemption behaves as it normally would without the debugger.

Currently, programs which use the preemptive tasking run–time and are to be run on a machine without a floating point co–processor must be linked with the software floating point library by specifying the `bamp -u` option at link time, even if the program does not use floating point operations explicitly itself.

Currently, the full benefits of task preemption are only available to programs running in Real Mode. Programs with the preemptive tasking run–time can be created to run in Extended Mode but they will have a lesser degree of preemption than the same program running in Real Mode. Basically, a task context switch may occur in such a program only at synchronization points and at places where task preemption is turned back on by the program or the run–time and NOT, in particular, in direct response to an external interrupt, such as might occur when a delay times out.

Since the DOS operating system and its associated BIOS are generally not reentrant, once a request is issued to DOS the request must be allowed to complete before another request is issued. With full preemption enabled, it would be possible for a task to gain control of the processor (via a hardware interrupt) while DOS was in the middle of handling a request from some other task. If this happens, the preempting task must not also issue a DOS request. This can be easily prevented by turning off preemption while a DOS request is in progress. As was mentioned before, the standard distribution library and associated run–time does disable preemption during such times. However, the *DOS Environment Library* does NOT currently protect itself, so the burden is on the user of this library to do so. In other words, if the application program uses the *DOS Environment Library* with the preemptive tasking run–time, then the program must surround every call to the DOS Environment Library with calls to `task_control` to disable preemption as illustrated in the following example.

```
task_control.pre_emption_off;
error := directory.make("c:\test.dir");
task_control.pre_emption_on;
```

6.3 Pragma Shared

Pragma **shared** is not necessary because of the way the compiler generates instructions on 80x86 processors. Pragma **shared** is ignored.

6.4 Interrupt Entries

Interrupt entries are supported. A task entry's address clause can be used to associate the entry with a PC–DOS interrupt. Values in the range **0 .. 255** are meaningful, and represent the interrupts corresponding to those values.

6.5 Memory Requirements

Each task has a private stack area. This stack area is allocated in the main program control stack (referenced by the SP register). In the 80x86 configuration, the total size of the program control stack is presently limited to 64K bytes (see Chapter 13). This limit on the total stack space available places an absolute restriction on the number of tasks that may be activated in a program.

Using the default stack allocations in Real Mode, the main program stack space is 20K bytes and the task stack space is 20K bytes, for a total of 40K bytes allocated for the entire program stack. In Extended Mode when tasking is used, the default stack allocations specify a main program stack of 44K bytes, and task stack space of 20K bytes, for a total of 64K bytes.

Note that no stack space is allocated for tasks when tasking is not used by a program. The total stack space allocated to all tasks may be changed by using the **–s** option to **bamp**. The total stack size allocated to the main program (that stack space allocated for use by everything but tasks) can be changed by using the **–M** option to **bamp**.

The amount of stack space required by a task is proportional to the number and sizes of local variables in the task and the depth of the task's call stack. The current individual task stack size defaults to 1024 bytes (1K). This places a limitation of about twenty activated tasks in the default stack allocation. The amount of stack space allocated for a task can be changed by using a length clause as in this example:

```
task type customer;  -- See previous example.
for customer'storage_size use 512;
  -- Length clause reserves 512 bytes for each
  -- instance of task type customer, as in:
hurried_customers: array(1..2) of customer;
  -- Each task component of hurried_customers has
  -- 512 bytes of stack space reserved for it.
```

In this example, the length clause specifies that half the usual amount of task stack space should be allocated for instances of task type **customer**.

Naturally, the less space reserved for each task stack, the greater the number of tasks that can be activated.

Note that implicit run–time support calls as well as explicit subprogram calls must be taken into account in deciding how large to make a task stack.

Chapter 7 Program Optimization

7.1 Purpose

The Meridian Ada Optimizer is an integrated component of the compiler. The Optimizer is used to improve the overall quality of the code output by the compiler in terms of both space and time. The Optimizer performs both local and global optimizations on your program by analyzing the internal representation of the program output by the compiler.

The Optimizer is not intended to replace good programming practices. It is often the case that algorithmic improvements result in excellent overall performance improvements; however, only you can understand the purpose and use of the program to the extent necessary to implement fundamental changes to the program's basic algorithms. The Optimizer has no such understanding and can only be used to improve the mapping of the program onto the target machine.

7.2 Operation of the Optimizer

To use the Optimizer, certain command line options are given to the **ada** and **bamp** commands. Examples are given below. Additional information is provided in sections 19.1 and 19.7.

1. During the compilation step(s), use the **ada -K** option to inform the compiler that the result of the compilation may be optimized, as in this example:

   ```
   ada -K prg_name.ada
   ```

   This option causes the compiler to keep information about the compilation available for later use by the Optimizer. If this option is not specified for a given compilation unit, then that compilation unit will not be optimized. The Optimizer displays a message to that effect during the link step. The un–optimized module should be re–compiled using this option. To use the Optimizer to its fullest extent, all compilation units in a program should be compiled with the **ada -K** option. Note the capitalization of the **-K** option.

   Note that extra disk space is required to store the information generated by the compilations.

2. Once the program has been compiled, the final step is to invoke **bamp** using one of the optimization options as in this example:

   ```
   bamp -G prg_name
   ```

   The **-G** option causes **bamp** to perform both local and global optimization for the program. To perform global optimization only, use the **bamp -g** option instead. Note that the alphabetic case of these options is significant, e.g. **-g** versus **-G**.

When optimization is requested and the **bamp** verbose option **-v** has been specified, information about the optimization process is printed during the final link step. If a message is printed that indicates a compilation unit has not been optimized but optimization was intended for that compilation unit, then be sure to recompile the unit with the **ada -K** option as discussed above.

7.3 When to Optimize

Since extra time is required during the link step to optimize a program, you need to make some decisions about development time tradeoffs. During the early and middle phases of a development cycle, it is probably not

worthwhile to optimize the program. The tradeoff will generally fall in favor of overall programmer productivity, which means that use of the Debugger and faster compile times will be more important than size or speed of your program. Once a project is nearing completion, however, it is appropriate to invoke the Optimizer to begin performance testing and integration. The Optimizer can also provide some information about your program that may be of interest, such as names of unused subprograms.

Note that using the **ada -K** option does not incur any additional compilation time overhead, although disk space overhead is incurred. In terms of compilation time, however, it doesn't hurt to use the **ada -K** option in preparation for use of the Optimizer.

7.4 Global Optimization

Global optimization consists of removing unused subprograms and global data objects from the final executable program image. This includes unused runtime code as well as program code. For example, if **text_io.put_line** is called by your program and no other **text_io** routines are called, then the remaining run–time routines that are not required are removed from the final program.

7.5 About .int Files

The **ada -K** option preserves a file with the extension **.int** for each library unit. This file is produced by the compiler. The file is deleted after code generation unless the **-K** option is given.

An **.int** file contains the Meridian Internal Form (MIF) representation of a library unit. MIF is a proprietary intermediate program representation generated by Meridian's Ada, Pascal, and C compilers for use by Meridian's code generators and optimization tools.

Chapter 8 Library Management

This chapter describes how the library management system works. An understanding of the details of library organization is required only for medium to large scale programming projects. In most small programming projects, nothing more than the **newlib** command is required for library management.

The library system simplifies many aspects of software development by:

- Enabling the compiler to check interrelationships among compilation units, including order of compilation.
- Relating file names and compilation units.
- Maintaining correspondences between symbol names assigned by the programmer and symbol names assigned by the compiler.
- Keeping track of what object modules to link into a program.

A software development project can be organized so that compilation units are spread out among several directories. Each directory contains a program library that maintains information about the compilation units in that directory. A program library (usually just called a *library*) also contains links to other libraries. These links simplify library version control and largely eliminate duplication of commonly used compilation units such as **text_io**. Links among libraries can form complex graph structures, and such structures should be organized carefully. Library links are discussed in sections 8.3 and 8.5.1.

8.1 The Library as a Database

A program library encompasses many aspects of a database management system. A database is managed by several functions:

- Create and initialize the database.
- Insert a new object into the database.
- Delete an object from the database.
- Modify an object currently in the database.

The compiler is one component in a code library management system, combining the functions needed to insert a new object into the database and to modify an object currently in the database. An "object" in this case is information about a compilation unit, in particular what its name is and what files are associated with it, along with other information. Whenever the compiler processes a compilation unit, either a new program library entry is made or an existing entry is modified for that compilation unit. An example of the contents of a program library (as listed by using **lslib -k**) is:

```
Links to:

c:\ada\adautil\ada.lib
c:\ada\paclib\ada.lib

Subprogram file_form_demo
Package io_example
Subprogram reschedule_demo
Subprogram copy_input_to_output
```

The `lslib` program examines a program library and prints a summary of the contents. This example shows links to other libraries and entries for several library units. The library unit entries are produced by compiling source files. Library links are discussed in more detail in sections 8.3 and 8.5.1.

More extensive information about a library and its entries can be produced by using the appropriate `lslib` options. An example of information maintained in a library about a particular compilation unit (as listed by using the `-l` option to `lslib`) is:

```
Subprogram copy_input_to_output
  Source file name is "copy_inp.ada".
  Internal link name is "aae".
  Host system file name is "copy_inp".
  Can be main program.
  Entered on Tue Apr 14 1987 11:45:51 PST.
  Last changed on Tue Apr 14 1987 11:50:37 PST.
  Dependent on: text_io
  Body is defined.
    Dependent on: text_io
```

The remaining database management functions are implemented by separate, complementary tools:

- The `mklib` command creates and initializes new program libraries.
- The `lnlib` command adds or deletes additional library references (links) to a program library.
- The `rmlib` command deletes compilation unit entries from a program library.

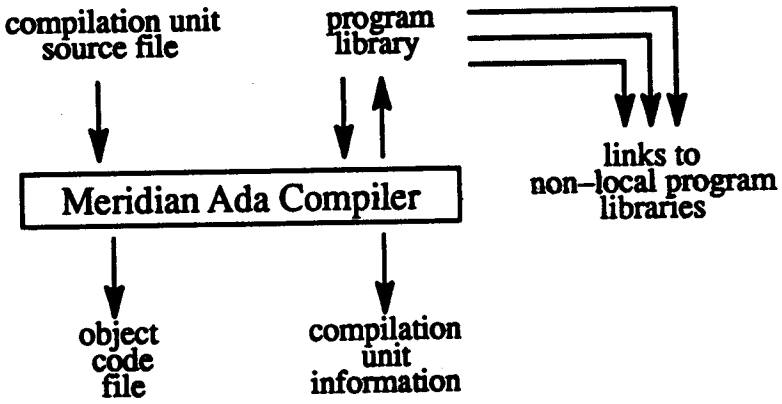


Figure 8.1 Compiler/Library Interaction

The `newlib` command simply invokes the `mklib` and `lnlib` commands with some useful defaults. Refer to Chapter 19 for details about these commands.

8.2 Compiler/Library Interaction

The interaction between the compiler and a library is shown in Figure 8.1. When a source file is compiled, the library is updated with information about any compilation units or subunits in the source file. In addition, the compiler produces object code files and interface description files, the names of which are derived from information recorded in the corresponding library entries. If the compiler encounters a context clause (a with clause) the library is consulted about the library unit named by the context clause.

Consider this compilation unit, a top–level instantiation of a generic package:

```
with direct_io;
package direct_int_io is new direct_io(integer);
```

When the compiler encounters the **with direct_io** clause, the compiler searches the library structure for **direct_io**, a standard generic package described in the LRM (also see Chapter 4 in this document). Under normal circumstances, **direct_io** is found in the standard library, **paclib\ada.lib**, a direct link to which must be present in the program library (the library in the current working directory) in order for the search to succeed. This link is created as a matter of course by the **newlib** command, which is invoked once in each directory in which compilations are to take place. Library links are discussed in more detail in sections 8.3 and 8.5.1. When the sample package **direct_int_io** is compiled producing an object code file and an interface description file, the compiler updates the program library with information about the compilation unit **direct_int_io**:

```
Package direct_int_io
  Source file name is "diio.ada".
  Internal link name is "aaf".
  Host system file name is "aaaaaaab".
  Entered on Tue Apr 14 1987 18:21:03 PST.
  Last changed on Tue Apr 14 1987 18:21:03 PST.
  No spec initialization
  Dependent on: direct_io
  Body is defined.
    No body initialization
```

The program library keeps track of how up–to–date a particular unit is with respect to others on which it depends. For example, if unit **a** withs unit **b**, but then in the course of re–compiling and refining the compilation units of the overall program, something is added to or deleted from **b**'s specification, then unit **a** must be re–compiled. As soon as **b** is re–compiled with the changed specification, the library entry for **a** is marked as obsolete.

Another circumstance under which entries for library units can be marked as obsolete is when a "local" unit replaces a "remote" unit with the same name. For example, if you were to write and compile your own version of **text_io**, then any entries for units in the program library that formerly depended on the distribution version of **text_io** would be marked as obsolete, requiring that they be re–compiled before they can be used in other programs.

8.3 Library Links

After a program library is created using **mklib**, link entries to other program libraries can be given using **lnlib** to make additional compilation units available. This affords the advantage that all compilation units need not be compiled in the same directory, thus making it easier to maintain version–controlled archives of tested compilation units. Libraries with standard setups are created with the **newlib** command, which simply invokes the **mklib** and **lnlib** commands with the appropriate defaults.

When a context clause (a **with** clause) is encountered, the compiler searches two levels of the library structure: the program library and the libraries to which there are direct links from the program library. In other words, the compiler only "sees" those library entries that have been immediately linked with the program library, whereas further links and therefore further dependencies are not immediately visible. This scheme is directly analogous to the way context clauses work: an entire hierarchy of dependencies is not made immediately visible by a single context clause; only those objects visible in the explicitly named compilation unit are available. For example, if **a** withs **b**, and **b** withs **c**, then **a** can use **b** directly, but not **c** (unless **a** also withs **c**). A more detailed example is given below.

```
with c;
package b is
...
end b;
-----------------
with b, c;
procedure a is
...
end a;
-----------------
package c is
...
end c;
```

Figure 8.2 Sample Library Units a, b, c

Additional levels of indirection in the library structure are used by the compiler and by the **bamp** program to maintain unique symbol names across all compilation units of a program (see section 8.5.1). The library links are also used by the **bamp** program to determine which compilation units are to be bound into a particular program. To provide an example of how the compiler's library search works, refer to Figure 8.2, Figure 8.3, Figure 8.4, and Figure 8.5. The latter three figures show different possible library arrangements for three compilation units, **a**, **b**, and **c**. Partial definitions for **a**, **b**, and **c** are given in Figure Figure 8.2. Note that **b** also depends on **c**. To simplify the discourse, it is said that the program library is the library that contains, or will contain following compilation, the entry for **a**.

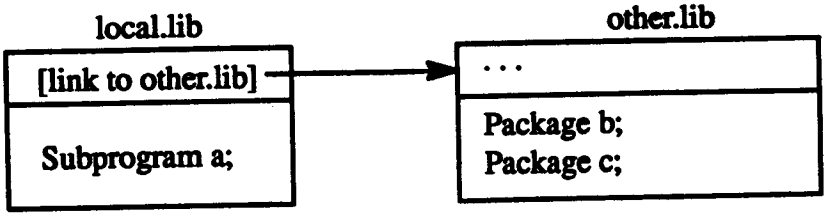


Figure 8.3 b and c Immediately Available to a

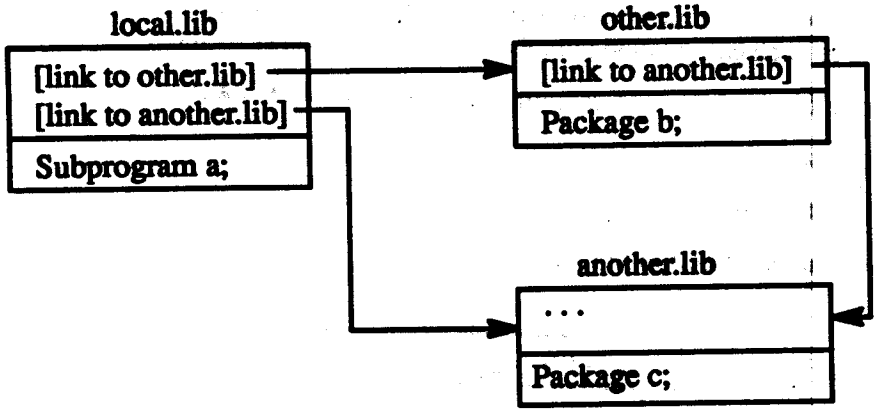


Figure 8.4 b and c Immediately Available to a

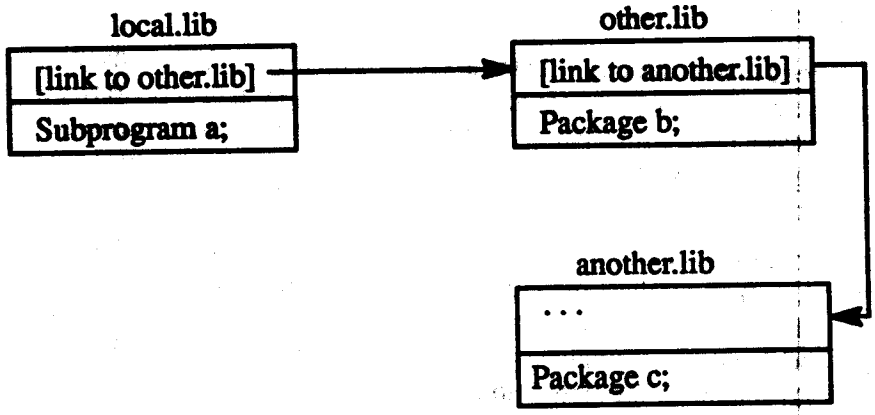


Figure 8.5 c Not Immediately Available to a

In the library arrangement shown in Figure 8.3, both b and c have entries in the same library, and this library is referenced by a link entry in the program library. This makes it possible to specify context clauses for both b and c in the compilation of a.

In the arrangement shown in Figure 8.4, entries for b and c exist in separate libraries, and both libraries are referenced by link entries in the program library. Again, this makes it possible to specify context clauses for both b and c in a.

In the arrangement shown in Figure 8.5, entries for b and c exist in separate libraries, but only the library containing the entry for b is referenced by a link entry in the program library. Although there is a further link entry in the library containing the entry for b to the library containing the entry for c, this is insufficient to make c available to a: the compiler reports an error in attempting to find the information it needs for the compilation unit c.

8.4 Auxiliary Directories

When a source file is compiled, a number of associated files are created depending on what kind of compilation units are involved. For each compilation unit, the associated files may include:

- an object code file (`.obj`) or an assembly language code file (`.asm`)
- an interface description file (`.atr`)
- generic description files (`.gnn`)
- subunit environment description files (`.sep`)

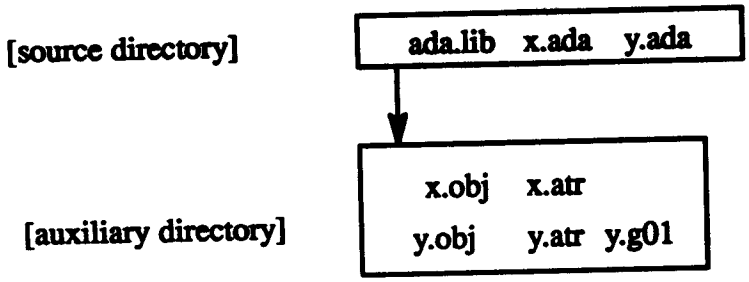


Figure 8.6 Auxiliary Directory Arrangement

Since you usually never need to deal directly with the associated files, it can be inconvenient or unsightly to keep these files in the same directory with the source files. The library system supports *auxiliary directories* in which the various files associated with a particular library's compilation units are kept. This kind of arrangement is shown in Figure 8.6. Without an auxiliary directory, all associated files are kept in the same directory in which the compilation takes place.

The `newlib` command builds an auxiliary directory by default. If the `mklib` command is used in place of the `newlib` command, then an auxiliary directory, if one is desired, must be specified with the `-a` option to the `mklib` command. The `mklib` command does not automatically create the auxiliary directory; it must be created with the PC-DOS `mkdir` command prior to the first compilation involving the library. Refer to the details of the `mklib` command for more information.

8.5 Special Considerations

This section discusses some of the idiosyncrasies of the Meridian Ada library management system. You should be aware of these aspects so that recovery from certain unusual error conditions is made easier.

8.5.1 Unique Names and Library Links

Because the lengths of programmer defined Ada identifiers may exceed limitations for file names and linker symbol names, somewhat shorter, unique abbreviations or "artificial" names are generated and name mappings recorded in the library.

The unique file name mappings used to prevent collisions among similarly named library units are coordinated among libraries that are linked together. These mappings are not coordinated among disjoint libraries. This has several implications. First, this means that in most cases, multiple program libraries should not coexist in the same directory because of file name collisions. It is possible to maintain co-resident program libraries, each library with its own auxiliary directory. This permits any number of program libraries to be co-

resident. To maintain such an arrangement, more work is required than simply using a single program library with the default name in each directory, since the non–default library must then be named explicitly with each library management command (including compilation). Auxiliary directories are discussed in section 8.4.

Next, you must be aware that certain library arrangements pose interesting problems when the time comes to merge sections of separately maintained program development efforts. Some arrangements prevent the library system from properly maintaining unique names across a set of compilation units. Consider the arrangement shown in Figure 8.7. The disjoint libraries X, Y, and Z all have link entries for the library A. Unique name mappings are not coordinated among X, Y, and Z, although the mappings are coordinated between A and any of the individual libraries. Note that the lack of coordination among X, Y, and Z only poses a problem if modules in X, Y, and Z are ultimately linked together.

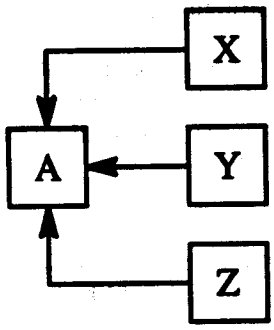


Figure 8.7 Disjoint Libraries

In Figure 8.8, another problematic arrangement is shown. In this case, unique mappings are not coordinated between libraries A and B, creating possible complications for compilation units that use library Z. Note that the lack of coordination between A and B is not a problem if both libraries are "frozen".

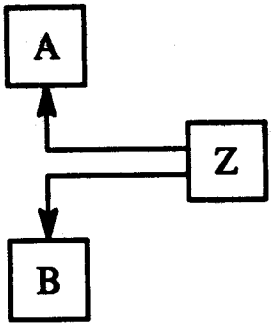


Figure 8.8 Another Disjoint Library Arrangement

One possible way around such problems is to make additional library links, provided that the links are acyclic, as shown in Figure 8.9.

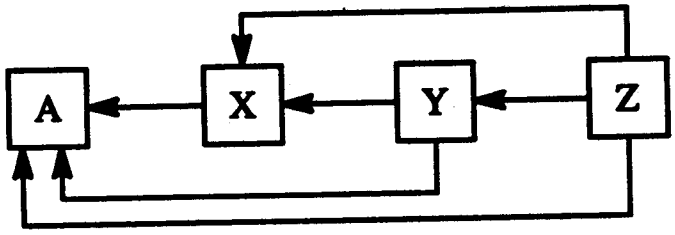


Figure 8.9 Fully Connected Library Structure

An important point for programming project library management is that in any given subtree of a library graph, the libraries lower in the tree should be changed relatively infrequently. In Figure 8.7, library A should be changed as little as possible, if libraries X, Y, and Z are the libraries for the main code development efforts.

8.5.2 Library Integrity

Terminal–originated interrupts (Control–C (^C)) are disabled during the short periods that library updates are in progress. This is to prevent possible corruptions of library contents.

The library management tools provide ways to handle the associated files automatically, but some planning is required when deleting or moving files. If files are to be moved or deleted, the `rmlib` program should be used to update the library. Some recompilations may be necessary as well.

The `rmlib` command removes entries from a program library. Source files are left intact, but associated files are deleted. Since an individual library entry may be part of a complex dependency graph, a number of other related library entries may have to be deleted first to maintain the integrity of the graph. If a unit is removed from a library, other units that depend on the deleted unit cannot be compiled.

In the unlikely event that a library database file becomes corrupt (usually because a library update was somehow aborted in midstream) it may be necessary to delete the program library. The library must then be reconstructed from scratch and the entries restored by recompiling all the source files in the directory. Of course, if an archival backup copy of the program library is available, then only the units modified since the backup operation need be compiled once the file is restored.

Chapter 9 Floating Point Software

The Meridian Ada Software Floating Point Run-Time (SFPRT) permits an Meridian Ada program to run on systems without an 80x87 math co-processor (an 8087, 80287, or 80387), but takes full advantage of a math co-processor if installed. The information in this section applies only to those programs containing floating point computations. Programs that use only fixed point or integer computations are not affected.

To ensure that programs operate safely on any system, a missing math co-processor causes a **program_error** exception to be raised with the message "**must install math coprocessor to use float operations**" unless SFPRT is used. SFPRT is linked with a program whenever the **bamp -u** option is used. To ensure that a program containing floating point operations will run on all systems regardless of the presence or absence of a math co-processor, the program should be linked with the **bamp -u** option.

To permit programs to run more quickly on systems with math co-processors, the compiler always generates 80x87 instructions for floating point operations. SFPRT, if used, interprets the 80x87 instructions. SFPRT is used only if no 80x87 is present. If present, the math co-processor is always used for floating point operations, regardless of whether or not a program is linked with SFPRT.

By default, a call to a special routine is placed before each sequence of floating point instructions in the generated machine code. The special routine takes action based on whether or not a math co-processor is installed and whether or not the floating point software is present. If there is no math co-processor available, the routine either raises the **program_error** exception or uses SFPRT, if SFPRT was linked into the program. The **ada -fF** option suppresses this call. This makes a program smaller and faster, but by using this option, the programmer "guarantees" that the program so compiled will never be used on a system without a math co-processor, since the normal checks are eliminated and SFPRT then cannot be used. If a program compiled with **-fF** is run on a machine with no math co-processor, the machine may simply "freeze up" in this circumstance, requiring a reboot.

The **ada -fF** option and the **bamp -u** option should not be used in the same program; it is pointless to link SFPRT into a program when all calls to SFPRT have been suppressed by **-fF**.

The following table summarizes the various combinations of options and their effects on program execution depending on whether or not an 80x87 math co-processor is present in a particular machine. The cases where SFPRT is available are emphasized, since this is the most robust program configuration.

ada -fF	bamp -u	80x87 present at run time	effect
No	No	No	Program fails: **program_error** exception raised; no SFPRT available
No	No	Yes	Program must and does use 80x87; no SFPRT available
No	Yes	No	Program can and does use SFPRT
No	Yes	Yes	Program can and does use SFPRT;
Yes	No	No	Program fails; behavior unpredictable; checks for co-processor suppressed; no SFPRT available

ada –fF	bamp –u	80x87 present at run time	effect
Yes	No	Yes	Program must and does use 80x87; checks for co–processor suppressed; no SFPRT available; program smaller and faster
Yes	Yes	No	not recommended; Program fails, even though SFPRT is present; calls to SFPRT suppressed; behavior unpredictable
Yes	Yes	Yes	not recommended; Program must and does use 80x87; SFPRT present, but not used; calls to SFPRT suppressed

Note: There are some very minor differences in precision between results computed by SFPRT and the 80x87 math co–processor. These differences are most noticeable in the results of transcendental functions from package `math_lib` in the Meridian Ada Utility Library, although the differences are still small.

Chapter 10 Using the Debugger

10.1 Introduction

The Meridian Ada Debugger is an interactive source–level debugger for use with programs written using the Meridian Ada compiler. The Debugger allows you to debug in high–level Ada terms; no knowledge of the underlying machine architecture is required. A debugger command can be an Ada variable expression to be printed, an Ada assignment statement, or an Ada subprogram call (including Debugger subprogram calls).

The Debugger possesses an identifier completion facility, a command history mechanism, a command editing facility, and a help facility. The Meridian Ada Debugger can greatly improve productivity.

10.1.1 Overview of Debugger Facilities

The Meridian Ada Debugger command processor provides these facilities:

- simple commands with Ada syntax
- one–key identifier completion
- command history and editing
- on–line help

Debugger facilities for control of execution include the following features:

- source line breakpoints
- single stepping
- variable "watches"
- automatic exception breakpoint
- task breakpoints
- Debugger "at" commands

The program state can be examined in several ways:

- source code review
- object display
- subprogram call back trace
- active execution tracing

The program state can be modified at breakpoints with:

- variable assignments
- user subprogram calls

10.1.2 Source–Level versus Machine–Level

The Meridian Ada Debugger deals with programs in high–level Ada terms; no knowledge of the underlying machine architecture is required. In a traditional machine–level Debugger, you must be intimately familiar with the compiler, because you must set breakpoints on machine instructions, and values of variables are determined by dumping regions of memory. With the Meridian Ada Debugger, you set breakpoints on Ada statements, not the underlying machine code instructions, and the value of a variable or component can be easily printed using a format suitable for its type.

10.2 Debugger Tutorial

This tutorial introduces you to some of the features of the debugger. The program, **dbtest**, used during this exercise is located in the **test** directory where you installed the compiler.

Commands are executed by pressing ENTER*. For example, you would type the command and then press ENTER to execute. This instruction has been omitted from each step in order to avoid repetition and improve readability.

If you are using ACE, you can perform the debugger tutorial located in the *Meridian ACE User's Guide*.

If at any time you want to exit the debug session, type **quit** at the prompt (>).

1. Change to the **test** directory located in the directory where you installed the compiler.
2. If you have not already created a Meridian Ada library, do so now using the **newlib** command.

   ```
   newlib
   ```

3. Compile the program, **dbtest**. To prepare the program for debugging, use the **-fD** option.

   ```
   ada -fD dbtest.ada
   ```

4. Link the program using **bamp**.

   ```
   bamp dbtest
   ```

   The program has now been compiled and linked with debugging enabled.

5. Execute the program by typing the program name, **dbtest**. The program begins execution in the debugger, suspended at the start of the main program.

   ```
   dbtest
   ```

6. To get a listing of debugger commands, type **help**.

   ```
   help
   ```

   A listing of the available debugger commands is shown.

7. To print a listing of all local variables and their values use the **dumpall** command as shown below:

   ```
   dumpall
   ```

   The first line contains the subprogram name (**dbtest**) and the line at which the program was suspended (1). The values of all global variables in all units are displayed.

8. You can also examine the variables and their value by naming them directly. To display the value of the variable *x*, type the variable name at the prompt as in this example:

   ```
   x
   ```

   The current value of *x* is 0.

   Now display the value of *a* by typing the variable name *a* at the prompt:

   ```
   a
   ```

   The current value of *a* is **(RED..BLUE => 0)**.

9. To set a breakpoint on line 21 of **dbtest**, use the following command:

   ```
   breakp(dbtest,21)
   ```

This is also called the Return key on some systems.

Set breakpoints at lines 22 and 23.

```
breakp(dbtest,22)
```

```
breakp(dbtest,23)
```

10. To see the breakpoints, use the `list` command:

    ```
    list
    ```

    The command displays the program name and the line number where the breakpoints occur. The `list` command also indicates that there are no watches or traces in effect.

11. Issue the `ss` command. The `ss` command means "single step" through the program. The `ss` command also single steps into any calls.

    ```
    ss
    ```

    This executes to the first breakpoint.

12. If you now execute a `dumpall` command, you will be able to see how the program has executed through the initializations and how the variables were updated.

    ```
    dumpall
    ```

13. If you want to monitor a particular variable, use the `watch` command. For example, to monitor the variable `t`, enter the following command:

    ```
    watch(t)
    ```

14. Do a `list` command. Now you can see that the variable `t` has been added to the list under the `watches` section. The `watch` command monitors the stated variable (in this case, variable `t`) during execution, suspends the program, and displays the original value and current value when the variable is changed.

    ```
    list
    ```

15. Execute the program until `t` is changed by using the `c` (continue) command. The `c` command executes to the next set breakpoint.

    ```
    c
    ```

    Information about the watched variable, `t`, is displayed along with information about the next breakpoint.

16. To cancel the watch on variable `t`, issue an `unwatch` command as shown below:

    ```
    unwatch(t)
    ```

17. To see your position during the execution of a program you can use the `where` command. Enter the `where` command now.

    ```
    where
    ```

    The where command displays a listing of the current source statement including 5 lines before and 5 line after. The current line is denoted with an asterisk (*).

18. The `trace` command allows you to trace subprograms. To trace all subprograms, use the `traceall` command.

    ```
    traceall
    ```

    The traceall command traces through all subprograms. If you only want to trace through one subprogram use the `trace` (*subp*) command, substituting the subprogram in parentheses.

19. You can continue execution of the program using the continue command as shown below:

```
c
```

The output displays the call to procedure `p`, the calling value, the returning value of `p`, and procedure `p` returning.

20. To turn off a `trace`, use the `untraceall` command

```
untraceall
```

21. To continue through the program, issue another `c`(ontinue) command.

```
c
```

This should take you to the end of the program and the following messages are displayed:

```
wxyz
Program terminates normally.
```

22. To exit the debugger, use the `quit` command.

10.3 Preparing Code to be Debugged

The Debugger is not a separate program; instead, the debugger is linked directly into a user program. Compilation units to be debugged are compiled with the debugging flag `-fD` as in this example:

```
ada -fD test.ada
bamp test
```

When a program compiled in this way is linked and executed, the Meridian Ada Debugger is automatically invoked.

For compilation units compiled with debugging enabled, the compiler generates special debugging code with the compiled code. The debugging code makes calls to debugger library routines at strategic points, usually after every source statement line. The debugger library routines monitor execution of code within the debugged units. Any or all of the units in a program can be compiled with the debugging flag, although it is recommended that the main subprogram always be compiled for debugging. Those units compiled without debugging are inaccessible from the debugger. When `bamp` is run, the debugging run–time support is automatically linked into the program.

This code–insertion method allows for an extremely powerful debugger that can interact with your program to a high degree, but it does incur some additional overhead. Programs compiled in Debug mode tend to be somewhat larger and slower than those compiled normally.

10.4 Debugger Commands

When a program is compiled with debugging enabled, the program normally begins execution in the Debugger with the program suspended at the start of the main program. You can type commands to set and delete breakpoints, examine memory, get a stack trace, resume execution, or simply exit the Debugger.

Meridian Ada Debugger commands have several forms:

- an Ada variable reference
- an assignment
- a user subprogram call
- a Debugger call

These command forms are explained in the following sections.

If you have named a local variable using the same name as a debugger command, then typing the command displays the value of the variable. This means that the debugger command does not execute. To execute the debugger command instead of displaying the variable value, type:

debug. [*command_name*]

10.4.1 Ada Variable References

The value of any Ada object can be printed by simply typing the name of the object. If a simple object name is typed, the entire object is printed in a format appropriate for its type; numeric types as numbers, enumeration types as enumeration identifiers, strings as string literals, and arrays or records, or any combination thereof, as properly indented aggregates. Components of composite types can be printed by specifying the corresponding Ada name. The name can include array indexing or slicing, record component selection, **.all** selection for access types, and Ada attributes: **'first**, **'last**, **'length**, **'constrained**, **'address**, or **'size**. Examples are given below. Many other Debuggers have more complicated syntax for examining values at machine addresses, and permit only one level of indirection. The Meridian Ada Debugger has no such limitations.

Simple Objects

Simple objects are printed according to their underlying types as in this example:

```
x: integer := 12;
c: character := 'T';
b: Boolean := false;
```

These objects are printed simply by entering their names:

```
> x
12
> c
'T'
> b
FALSE
```

Array Objects

Consider these source declarations:

```
type color is (red, yellow, blue);
a: array(color) of integer := (2, 3, 5);
t: string(3..6) := "wxyz";
type rec is
  record
    x,y,z: integer;
  end record;
r: array(1..3) of rec :=  ((11,12,13),(21,22,23),(31,32,33));
type ptr is access rec;
p: ptr := new rec'(r(2));
```

When an array object name is typed, the array is printed as an aggregate using named notation:

```
> a
(RED => 2, YELLOW => 3, BLUE => 5)
```

The array can be indexed or sliced as in Ada:

```
> a(yellow..blue)
(YELLOW => 3, BLUE => 5)
> a(yellow)
3
```

Arrays of type string are printed specially:

```
> t
"wxyz"
> t(4..5)
"xy"
>t(6)
'z'
```

Array attributes can be used as in Ada:

```
> a'first
RED
> a'last
BLUE
> a'length
3
> t'first
3
> t'last
6
> t'length
4
```

Indexes and slices can use attributes as with normal Ada syntax:

```
> t(a'length)
'w'
> t(4..5)'length
2
```

Record Objects

Record objects are also printed as named aggregates. Complicated nested structures are printed in an indented, readable format:

```
> r
(
1 => (x => 11, y => 12, z => 13),
2 => (x => 21, y => 22, z => 23),
3 => (x => 31, y => 32, z => 33))
> r(2)
(x => 21, y => 22, z => 23)
> r(2).x
21
> r(2).z
23
```

Access Objects

The value of an access object is printed as an address in hexadecimal. The referenced object can be printed using Ada's `.all` construct:

```
> p
[4FFE:4FE6]
> p.all
> p.x
21
```

A pointer value consists of a 16–bit paragraph address and a 16–bit offset in Real Mode programs. A pointer value consists of a 16–bit segment selector and 16–bit offset in Extended Mode.

In most instances, a pointer value is examined only to see if it matches a second value. Its actual value is of no interest.

Refer to the Intel 80x86 literature for details about the machine–level addressing methods.

10.4.2 Output Formats

The following table summarizes the printing methods used for various Ada types:

Type	Output Format
integer	integer literal
enumeration	enumeration IMAGE
float	exponential real notation
fixed	real with correct precision
string	string literal
array	array aggregate
record	record aggregate
access	hex address value
task	task information
derived	use parent type
private	use full type
subtype	use base type

10.4.3 Subprogram Calls

Subprograms defined by you can be called directly from the Debugger. All necessary parameters must be provided. The return results of functions are printed according to their return value types. Subprogram calls and variable references are subject to visibility constraints (discussed in Section 10.13). Currently, only positional parameters are supported; named parameter notation cannot be used, and all default parameters must be specified explicitly. Debugger commands are simply built–in subprograms that can be called just like user subprograms, but some Debugger commands are treated specially in that they accept a variable number of parameters. All built–in Debugger subprograms are summarized in Section 10.16.

The calling facility allows you to test program components interactively, easily, and thoroughly without writing tedious test programs. It is also useful for displaying complex data structures such as trees or linked lists. You can define printing routines that can be called as needed from the Debugger.

Consider the following simple table lookup package:

```
package table is
  type element is
    record
      key: integer;
      value: character;
    end;
  procedure enter(key: integer; value: character);
  function lookup(key: integer) return character;
end table;
```

Suppose the body of this package allocates an `element` record whenever `enter` is called. The function `lookup` returns the entered value, if found, and `'*'` otherwise.

Once this package is compiled and linked with an empty main program, it can be easily tested using the interactive call facility of the Debugger:

```
> lookup(5)
'*'
> enter(3, 'a')
> enter(5, 'b')
> lookup(3)
'a'
> lookup(5)
'b'
> lookup(4)
'*'
```

If problems are found, they can be easily identified, and testing can continue more thoroughly in those areas.

An *in* parameter to a subprogram call in a Debugger command can be any legal Debugger expression. As always, only variable references can be given for *out* or *in out* parameters. Parameter lists (where required) are given as in Ada source code: enclosed in parentheses; each parameter separated from any others by commas.

Subprograms that have side effects will, when called from the Debugger, have the same side effects that they would have under normal run time conditions (e.g. `dispense_money`, `read_next_record`, `launch_missile`). Some caution is required when calling such subprograms in the Debugger.

10.4.4 Assignments

Assignments to variables are permitted using the normal Ada assignment notation, ":=". Any variable reference is permitted on both the left and right sides of an assignment:

```
> t
"wxyz"
> t(5) := 'A';
> t
"wxAz"
> t(4) := t(5);
"wAAz"
```

Expressions in assignments can be variable references (as above), any function call, scalar constants (e.g. 3.14159, 512, true), or quoted character literals (e.g. 'c'). Aggregates and multipart expressions (using infix unary or binary operators such as "+", "-", or `not`) cannot be used in Debugger assignment commands at present.

Some type checking is performed, but it is not as rigorous as the compiler's. Note that a semicolon (";") at the end of an assignment is optional. It is not possible to set the value of a packed record field or a packed array element using Debugger assignments.

Consider these source declarations:

```
type color is (red, yellow, blue);
f, g: float;
hue: color;
b: boolean;
c: character;
```

Some examples of Debugger assignment commands using these variables are:

`f := 3.14159;`	Simple assignment to a **float** variable, **f**.
`g := sin(f);`	Call to user-defined function **sin**; value assigned to a **float** variable **g**.
`hue := blue;`	Simple assignment to an enumeration variable, **hue**.
`b := odd(i);`	Call to user-defined function **odd**; assignment to a **boolean** variable, **b**.
`c := 'Z';`	Assignment to a **character** variable, **c**.

10.4.5 Debugger Calls

The Debugger contains many commands – actually specially defined subprograms – that can be called to display or modify the debugged program and the debugging environment. For example, the procedure **stack** can be called to print a stack trace, and the special procedure **breakp** can be called to set a breakpoint. These special subprograms are described in detail in the sections that follow. The commands are summarized in Section 10.16.

10.5 Displaying the Environment

Several debugger procedures—commands—are pre-defined for displaying information about the program being debugged.

10.5.1 Printing Source Code

The Ada source code can be displayed without having to specify a source file or line number. A summary of the source code display commands to the Meridian Ada Debugger follows. A complete description of each command appears in section 10.16.

print	When the Debugger **print** command is invoked with an Ada subprogram as its argument, a portion of the subprogram is displayed. Relative line numbers (starting at 1) are displayed with the source so that locations for desired breakpoints can be specified. The **print** command prints 10 lines.
more	The **more** command can be used to print subsequent sets of 10 lines.
find	The **find** command can be used to search the current procedure for a particular text pattern.
source	The **source** command can be used to specify an alternate directory in which source files are to be found. Normally the current directory is used.
where	When a breakpoint occurs, the current source line is displayed. The **where** command prints a small section of the source code centered about the line where execution is currently suspended.

10.5.2 Displaying the Call Stack

The **stack** command can be used to print the call stack. Each call that hasn't yet returned is printed, starting with the most recent one. The calls are printed as Ada calls using named parameter notation to specify the parameter values. For each call, the relative line number in the caller from which the call occurred is printed. A complete description of the **stack** command appears in section 10.16.

10.5.3 Dumping Variable Values

The **dump** command can be used to print all of the variables for a given subprogram or package, allowing you to look over the current state of the environment. The **dumpall** command dumps every variable in the entire program. A complete description of the **dump** commands appear in section 10.16.

10.6 Control of Execution

When a *breakpoint* is encountered during program execution, the program is suspended and the debugger command processor is invoked. Breakpoints normally occur automatically at the beginning and end of program execution; these automatic breakpoints can be suppressed with the `silence` command.

Commands related to breakpoints follow. A complete description of each command appears in section 10.16.

`silence`	The `silence` command suppresses the automatic breakpoints at the beginning and end of program execution. This command is normally placed in the debugger initialization file.
`breakp`	A breakpoint can be set on any Ada statement by specifying a subprogram and relative line number with the `breakp` command.
`c`	To resume the program from a breakpoint, use the "continue" command, `c`.
`ss`	The "single–step" command, `ss`, can be used to execute a single statement before returning to the debugger command level.

10.6.1 Setting Breakpoints

The `breakp` command can be used to set a breakpoint on a particular subprogram or statement. The `breakp` command takes a subprogram name as its first parameter, and a line number (relative to the start of the subprogram) as its second parameter. If the line number is missing, the breakpoint is set at the start of the subprogram. Relative line numbers are displayed when a subprogram is printed using the `print` command. The special line number –1 refers to the end of the subprogram.

10.6.2 Resuming Execution

At a breakpoint, there are several ways to resume execution. The simplest is the continue command, `c`. This causes the program to resume execution until another breakpoint is encountered, or until termination.

The single–step command `ss` causes the program to execute a single statement before returning to the debugger. If the current statement contains a call, the single–step command causes a breakpoint to occur at the start of the called subprogram.

The `go` command continues to the next statement within the current subprogram (calls are skipped over). Multiple statements on a single line count as one `go` step, as in this example:

```
a := 7; b := 3;
```

A `go` command can be given an optional line number parameter. In this case, execution proceeds until that line number is reached, or until the end of the current subprogram is reached. If a second line number parameter is given, execution proceeds until either one of the specified lines is reached. This can be useful at an if statement if you are unsure which of the then or else branches will be executed. If an explicit breakpoint occurs during a `go` operation, the `go` is cancelled.

A breakpoint can be removed by using the `unbreakp` command with the same parameter sequence as the `breakp` command. The `list` command can be used to see what breakpoints are currently set.

10.6.3 Watching Variables

A "watch" can be set on a variable, causing a breakpoint to occur after any statement that modifies the value of the variable. This can be especially useful in tracking down subtle bugs where variables are changing unexpectedly. The `watch` command takes a variable name as its parameter.

For example, consider the following program:

```
-- watch demo
procedure p is
  n: integer := 37;
begin
  n := 37;
  n := 73;
end;
```

If you set a **watch** on the variable **n**, and then continue execution, a breakpoint occurs when the value of **n** changes:

```
> watch(n)
> c
variable n has changed
old value was 37
new value is 73
Breakpoint at line 5 in p (line 6 in watch.ada)
    20*  n := 73;
>
```

The breakpoint message prints old and new values of the watched variable and the line where the change occurred. The relative line number within the procedure and the absolute line number in the source file are printed.

The **unwatch** command removes a watch. A watch must be removed on a local variable before the enclosing procedure exits, and a watch on an allocated object must be removed before the object is deallocated. The **list** command displays the current watches.

10.7 Monitoring Execution

This section describes methods for monitoring execution of the program being debugged. The trace commands can be used to cause the debugger to output the call and return structure of the program, and the **at** command can be used to cause a specified action to occur whenever a particular line is executed.

10.7.1 Subprogram Tracing

Subprograms can be traced by using the **trace** or **traceall** commands. When a traced subprogram is called, the debugger writes out the name of the subprogram, and the names and values of its parameters. The **trace** command turns on tracing for a specific subprogram as in this example:

```
> trace(lookup)
```

Given this example command, whenever the user–defined subprogram **lookup** is called during program execution, the call is printed as an Ada call with named parameters. When the subprogram returns, the return value (if it is a function) is printed. The information is indented as the calls nest more deeply, and the indentation is properly maintained when exceptions disrupt the normal call and return structure of a program. The **traceall** command can be used to trace every subprogram in the program.

```
with ada_io;  use ada_io;
procedure factorial is
```

```
function fact(n: integer) return integer is
begin
  if n < 2 then
   return 1;
  else
   return n * fact(n - 1);
  end if;
end;

procedure do_fact(x: integer) is
  f: integer;
begin
  f := fact(x);
  put("fact(");
  put(x);
  put(") = ");
  put(f);
  new_line;
end;

begin
  do_fact(3);
end;
```

An example of a debugger session using the `traceall` command with the above program is shown below.

```
> traceall
everything traced
> c
do_fact(x=> 3)
  fact(n=> 3)
    fact(n=> 2)
      fact(n=> 1)
      fact returning 1
    fact returning 2
  fact returning 6
fact (3) = 6
do_fact returning
Program terminates normally.
> quit
```

Parameters and function return values are printed using Ada object type information. The `untrace` and `untraceall` commands are used to remove traces. The `list` command prints the current traces.

10.7.2 Inserting Commands

Debugger commands can be inserted into the code so that they execute each time a particular point in the code is reached. The `at` command specifies a subprogram and relative line number, and the command to be executed at that point. This allows you to insert `put` statements into the code dynamically, without having to recompile the program.

For example, for the factorial program above, we insert a command to print the value of `n` when `fact` is entered:

```
> at(fact, 3) n
> c
3
2
1
fact(3) = 6
Program terminates normally.
> quit
```

10.8 Conditional Breakpoints

The **when** and **stop** commands can be used in conjunction with the **at** command to set conditional breakpoints. The command:

```
> at(fun, 27) when(x > 37) stop
```

tells the debugger to check the value of **x** every time line 27 of subprogram **fun** is reached. If the value of **x** is greater than 37, a breakpoint occurs.

The conditional expression for **when** must be of a boolean type. The debugger supports Ada's pre–defined comparison operators: =, /=, <, <=, >, >=. No other operators are supported, but the expression can call a user–defined boolean function to test for arbitrarily complicated conditions.

For example, if you defined the following package:

```
package counter is
  n: integer := 0;

  function check(i: integer) return boolean;
end;

package body counter is
  function check(i: integer) return boolean is
  begin
    n := n + 1;      -- increment counter
    if i < n then
      return false; -- not there yet
    else
      n := 0;        -- reset  for  next  use
      return true;   -- this is the one
    end if;
  end
end;
```

and you entered the following debugger command:

```
> at(fun, 27) when(counter.check(10)) stop
```

then the program will reach a breakpoint on the tenth time that line 27 of subprogram **fun** is reached.

The action specified in a **when** command doesn't have to be a **stop** command. Any legal command can be specified.

For example the following line of code prints the value of **x** every time line 27 of subprogram **fun** is reached, if the value of **x** is greater than 37, but in either case, the program does not stop.

```
> at(fun, 27) when(x > 37) x
```

The **when** command is normally only useful inside of an **at** command. If a **when** command is entered at command level, its effect is that of an **if** statement. A **stop** command is normally only useful in a **when** statement. If a **stop** command is entered at command level, its effect is to make the next continue command be a single step (**ss**).

10.9 The Command Processor

The Meridian Ada Debugger includes a sophisticated command processor that is designed to minimize the amount of typing required to debug a program.

10.9.1 Identifier Completion

Most of the typing during debugging is done to specify identifiers for Ada subprograms or objects, or for the pre–defined debugger commands. The identifier completion facility eliminates much of this typing. Once enough characters have been typed to uniquely determine the identifier, a completion key (the ESC key) can be specified to automatically type the remaining characters. If the prefix does not uniquely determine an identifier, the terminal gives an audible "beep". In this case, you simply type more characters before trying completion again. A second completion key displays the possible completing identifiers.

For example, in the following command, the lower–case characters are typed by the programmer, and the upper–case characters are supplied by the command processor at points where the ESC key is pressed:

```
> datABASE(indEXVAL).comPONENT.nexTFIELD
```

Because of this facility, you are free to use long, expressive names for identifiers without incurring additional typing costs while debugging. This enhances the readability and maintainability of your programs.

10.9.2 Command History

The command processor also includes a history mechanism with an editing capability. You can retrieve previous commands which can be edited and re–executed. If a complicated data structure is under investigation, you can type several complex component references that differ only in small ways. The history and editing facilities can greatly speed up the typing of such commands. Also, when an incorrect expression is typed, the history and editing facilities can be used to quickly correct the error.

To retrieve the previous command, the Control– K (^K) or up arrow key is used. Each Control–K (^K) or up arrow key typed moves backwards one command. The Control–J (^J) or down arrow key is used to advance forward in the command history.

10.9.3 Command Editing

The editing facility includes commands to move left or right through the command one character or one identifier at a time and commands to insert or delete characters or identifiers. Editing commands are control characters, arrow keys, or other special keys such as DEL or END. The Control–Q (^Q) character (for question) can be typed at any time for editor help. The list of keys that are used in command editing are:

Key	Edit Function
Control–A (^A)	Insert a space
Control–F (^F)	Delete to end of line
Control–H (^H) or left arrow	Move cursor left one space
Control–J (^J) or down arrow	Next history command
Control–K (^K) or up arrow	Previous history command
Control–L (^L) or right arrow	Move cursor right one space
Control–N (^N)	Enable/Disable insert mode
Control–R (^R)	Redraw current line
Control–S (^S)	Delete current character
Control–T (^T)	Move right one word
Control–U (^U)	Delete entire line
Control–V (^V) or END	Move cursor to end of line
Control–W (^W)	Move left one word
Control–X (^X)	Delete current word
Control–Y (^Y) or HOME	Move cursor to start of line
DEL	Delete previous character
ENTER/RETURN	Execute current line
ESC	Complete current identifier

10.9.4 Function Keys

The most frequently–used commands that do not require parameters are associated with the function keys on the keyboard. Pressing the appropriate function key causes the corresponding command to be entered into the input buffer and executed. The function–key commands are:

Function Key	Command
F1	Help
F2	C
F3	SS
F4	Go
F5	Stack
F6	Where
F7	More
F8	Find
F9	List
F10	Quit

10.10 Special Commands

10.10.1 Start–up Commands

If the file **debug.ini** exists in the current working directory when the debugged program begins, the debugger reads commands from the file. This can be used to avoid having to set the same breakpoints many times while debugging a program.

It is also a good place to use the **silence** command, which causes the debugger to suppress the normal breakpoints at the start and end of a program. When this is done, the debugger is not heard from unless an execution error or breakpoint occurs.

The **debug.ini** file also often contains **source** commands for specifying the location of the program source code.

10.10.2 Redirecting Debugger I/O

The commands **set_input** and **set_output** can be used to redirect the debugger's input or output to a file or a different device. The program's I/O is unaffected. Each of these commands takes a single string parameter:

```
> set_output("results.txt")
> set_input("aux:")
```

10.10.3 Executing DOS Commands

The **execute** command can be used to execute a DOS command from within the debugger:

```
> execute("dir *.ada")
```

If no command is specified, a DOS command processor is invoked, allowing you to type many DOS commands. The DOS **exit** command returns to the debugger.

10.11 Exceptions

The Meridian Ada Debugger includes a facility for causing breakpoints to occur whenever an exception is raised, and for examining the state of the program after an exception has propagated out of the main procedure.

10.11.1 Breakpoints When Exceptions Are Raised

If an exception is raised by a program during a debugging session, it is propagated and handled normally. The command **excbreak** can be used to cause a debugger breakpoint whenever an exception is raised:

```
> excbreak(true)
```

A boolean parameter is required; true enables the feature, and false disables it.

Consider the following test program:

```
procedure excdemo is
  a: array(1..10) of integer;

  procedure p(n: integer) is
  begin
    a(n) := 37;
  end;
begin
  p(3);
  p(12);
end;
```

If we run this program under the debugger, with **excbreak** enabled, we reach a breakpoint when the exception is first raised:

```
> excbreak(true)
> c
Exception constraint_error raised
Breakpoint at line 3 in p (line 6 in excdemo.ada)
     3*  a(n) := 37;
> n
12
> a'first
1
> a'last
10
> stack
p(n => 12):3
excdemo():10
>
```

You can continue from such a breakpoint, but if there is no exception handler for the exception in the program, the exception propagates out of the main program, and the program terminates abnormally.

10.11.2 Exception Propagation

For programs that contain no exception handlers, it is always a good idea to enable **excbreak**. This can be done in the **debug.ini** file. If the program handles many exceptions, then **excbreak** may cause too many breakpoints during normal execution to be useful. If **excbreak** is not enabled, and an exception is propagated through the main program, a breakpoint is still reached. In this case, the debugger may be used to find the location where the exception was first raised, and the **stack** command shows the call chain, but the values of parameters and local variables may have been destroyed by the time the breakpoint is reached. To set a breakpoint when an exception is handled, simply use **breakp** to set a breakpoint in the exception handler.

10.12 Tasking

The Meridian Ada Debugger includes several features to facilitate the debugging of programs containing tasking. Breakpoints can be set in task bodies in the same way that they are set in subprograms, by specifying a task unit name and a relative line number. For task types, breakpoints that are set in the task body apply to any object of that task type.

10.12.1 Examining Task Objects

When an object of a task type is given as a command, it is printed in a format similar to an aggregate, containing pieces of information about the task. The information printed is:

- The name of the task object.
- The internal name of the task object.
- The execution state of the task.
- The source line where the task is currently executing.

10.12.2 Printing Task Names

In many situations, the debugger must print the name of a task, as in the output of the **where** command, but not all tasks may have names meaningful to you. The pre–defined procedure **settaskname** can be used to assign a more recognizable name to a task.

For a simple task object, the name of the task is automatically initialized by the tasking runtime. For task objects that are components of structures or dynamically allocated, the task name is not set. In this case, an internal name is used to identify the task. This name is generated by prefixing the character `'T'` to a unique task ordinal. Consider these declarations assuming `tsk` is a task type:

```
a: tsk;
b: array(1..2) of tsk;
```

In this example, the name of task `a` is "a", but the name of task `b(1)` is "T2". The pre–defined procedure `settaskname` may be used to assign a more recognizable name to a task:

```
> settaskname(b(2), "b2");
```

It should only be called for tasks that are not simple objects.

10.12.3 Execution States

A task may be in any one of the execution states listed below. The states should be self–explanatory given a sufficiently detailed understanding of standard Ada tasking.

- Inactive
- Being activated
- Currently running
- Ready to run
- Waiting for activation of dependents
- Waiting for delay
- Waiting for entry call to return
- Waiting for timed entry call
- Waiting at accept statement
- Waiting for rendezvous to complete
- Selective wait with timeout
- Selective wait with terminate
- Waiting for completion of dependents
- Aborted
- Completed
- Terminated

10.12.4 Breakpoints on Context Switches

You can request that a breakpoint occur whenever an Ada task context switch occurs. This is done by calling the `taskbreak` procedure:

```
> taskbreak(true)
```

When a Boolean parameter is supplied the following action occurs:

- `true` enables the feature and breakpointing can occur.
- `false` disables the feature and no break point occurs.

At the breakpoint, the names of the task being suspended and the task being resumed are printed. The debugger context is the context of the task being suspended. A single–step command, **ss**, advances the context to that of the task being resumed.

In some cases, the tasking runtime can detect a task deadlock. In this case, a debugger breakpoint is always reached, regardless of the status of **taskbreak**.

10.13 Visibility Issues

This section discusses visibility issues and qualification of variables or subprograms.

10.13.1 The Dynamic Call Chain

As mentioned in previous chapters, subprogram calls and variable references can be made from the debugger only subject to visibility constraints. This means that in any given debugging context, one must be able to "see" a variable or a subprogram to make use of it. The rules concerning what is or is not visible are slightly more complicated in debugger commands than they are under normal circumstances in Ada source programs. For example, suppose that a variable named **i** occurs as a local variable in several subprograms that call each other in succession:

```
procedure visdemo is

  procedure c is
    i: integer;
  begin
    null;
  end c;

  procedure b is
    i: character;
  begin
    c;
  end b;

  procedure a is
    i: integer;
  begin
    b;
  end a;
begin
  a;
end visdemo;
```

Procedure **a** calls **b**, which in turn calls **c**. If a breakpoint is reached in procedure **c**, then a reference to the variable **i** refers to the most recently seen instance of **i** (the one in **c**). To refer to a specific instance of a local variable **i**, the variable reference must be *qualified* with the name of the active procedure in which it occurs:

- **a.i** Refers to the instance of **i** in procedure **a**. Procedure **a** must be active, or a reference to **a.i** is not allowed.
- **b.i** Refers to the instance of **i** in procedure **b**. Procedure **b** must be active, or a reference to **b.i** is not allowed.
- **c.i** Refers to the instance of **i** in procedure **c** (this is the same as **i** in this example). Procedure **c** must be active, or a reference to **c.i** is not allowed.
- **i** Alone refers to the most recently seen name **i**. For example, if a breakpoint is triggered in **a**, then **i** refers to **a.i**. If a breakpoint is triggered in **b**, then **i** refers to **b.i**. Similarly, if a breakpoint is triggered in **c**, then **i** refers to **c.i**.

A procedure is *active* if it is part of the dynamic call chain, that is, any procedure displayed by the `stack` command.

The debugger treats parameters as local variables. Consider this example:

```
procedure myproc(a, b: integer) is
  c: character;
begin
  null;
end;
```

The parameters `a` and `b` may be referenced as `myproc.a` and `myproc.b`, just as the local variable `c` may be referenced as `myproc.c` – provided, of course, that procedure `myproc` is active.

10.13.2 Visibility and Scope

Objects local to subprograms (i.e. local variables and nested subprograms) may not be referenced in debugger commands unless they are visible. Subprograms and variables declared in the outermost part of a library package (i.e. *global* objects) are always visible. Local variables and nested subprograms are visible only if they are active. This definition of visibility provides somewhat more flexibility than the normal Ada static scoping rules. Consider this example:

```
procedure scopedemo is
  i: integer;

  procedure p1 is
  begin
    null;
  end p1;

  procedure p2 is
    i: integer;
  begin
    p1;
  end p2;
begin
  p2;
end scopedemo;
```

In this example, procedure `scopedemo` calls nested procedure `p2`, which in turn calls procedure `p1`. Procedure `p1` is at the same static scoping level as `p2`. If a breakpoint occurs in `p1`, references to `i` will refer to the `i` declared in `p2`. The Ada compiler resolves references to `i` within procedure `p1` to the `i` in `scopedemo`.

The Ada language uses "static scoping" and the Ada debugger uses "dynamic scoping". Dynamic scoping is more useful during debugging (because debugging focuses on the dynamic properties of a program), but you should be aware of this subtle difference in the visibility rules. Problems with dynamic *vs.* static scoping visibility issues should never arise in well–written Ada code.

10.13.3 Package Qualifiers

Variables and subprograms may be qualified with the names of the packages in which they occur as in Ada by using "dot notation". Consider this package:

```
package test is
  x, y: integer;
  procedure a;
end test;
```

```
package body test is
  p, q: boolean;

  procedure a is
  begin
    null;
  end;

  procedure b;
  begin
    null;
  end;
end;
```

A list of legal object qualifications is:

Name	Meaning
`test.x`	integer variable
`test.y`	integer variable
`test.a`	procedure
`test.p`	Boolean variable
`test.q`	Boolean variable
`test.b`	procedure

Note that packages are considered to be active in a top–down order (in reverse order of the elaboration of initialization code). This means that when there are collisions among the names of global objects, the last elaborated units win: un–qualified references to global objects refer to the "most recently seen" objects, just as with subprogram activations. Global objects in the main program are always most recent in the order of activation.

If a debugger command is hidden by a user–defined symbol, it may have to be qualified with the prefix `debug.` and some intrinsic subprograms may have to be qualified with the prefix `standard`.

For example:

`debug.breakp(test.b)`	set breakpoint at `test.b`.
`debug.c`	continue from breakpoint.
`q := standard.true`	patch the Boolean value of `q`.

These qualifications of `debug` and `standard` objects can be necessary because the pre–defined packages `debug` and `standard` are farther down in the dynamic call chain than user–defined units, thus user–defined subprogram or variable names may collide with debugger commands (for example, if `c` is a user–defined variable).

For purposes of debugging, you should refrain from defining an object, subprogram, package, or enumeration value with the name `debug` or `standard` and should also refrain from redefining the other debugger command identifiers.

10.14 Overloading

An Ada program may have many subprograms or enumeration literals with the same names. In this case, the name is said to be overloaded. The best way to distinguish between two distinct entities with the same name is through the use of "dot notation". For example, if a function `f` appears both in package `a` and package `b`, the function in package `a` could be displayed with this command:

```
> print(a.f)
```

In many cases, however, several overloaded occurrences of the same identifier may appear in the same package, e.g. `text_io.put`.

10.14.1 Identifying by Position

If a single package has more than one occurrence of a given identifier, dot notation is not sufficient to distinguish them. For this case, a special indexing construct is used to identify particular entities. For example, assume there are two different procedures `p` in package `g`:

```
package g is
  procedure p(x: integer);
  procedure p(y, z: integer);
end;
```

They can be accessed from the debugger by specifying a position value enclosed in brackets:

```
> print(g.p[1])
procedure p(x: integer) is
...
> breakp(g.p[2])
```

Since the position number is the position within a particular package, that package must be specified as part of the name using dot notation. Note that the bracket characters are not normally legal Ada characters.

The numeric position value is determined by the order in which the overloaded entities are defined. For a package, the definition order in the package specification is used (the order in the body could be different). In the above example, the first `p` in the package (the one with the single parameter `x`) is printed, and the second one has a breakpoint set on it. If there is any doubt about which position value a particular subprogram has, the `print` procedure should be used to print the header of that subprogram. If no position is specified, position 1 is assumed.

10.14.2 Calling an Overloaded Subprogram

When calling an overloaded subprogram interactively from the debugger, a position number must be specified:

```
> g.p[1](37)
> g.p[2](73, 99)
```

10.14.3 User–Defined Operators

User–defined operators are accessed from within the Meridian Ada Debugger as they are from Ada, by enclosing the name in double–quotes. For example, consider a single "`*`" function declared in a package `p`:

```
function "*"(a,b: complex) return complex;
```

This function may be accessed as expected from the debugger:

```
> breakp(p."*")
```

If a second `"*"` function is declared in the same package:

```
function "*"(a: complex; k: float) return complex;
```

then a position number must be specified:

```
> breakp(p."*"[1])
> print(p."*"[2])
function "*"(a: complex; k: float) return complex is
...
```

These operators may be called from the debugger using prefix notation only:

```
> p."*"[1](x,y)
```

10.14.4 Recursive Occurrences of the Same Identifier

The position number can also be used to distinguish dynamic occurrences of the same variable (see section 10.13). Consider the following function:

```
> print(fact)
1 function fact(n: integer) is
2 begin
3   if n = 0 then
4     return 1;
5   else
6     return n * fact(n-1);
7 end;
```

If we set a breakpoint at line 4 of `fact` (the "return 1"), and call the function with a value of 3, then when we reach the breakpoint, there are four distinct invocations of the function `fact`, as shown by this call back-trace:

```
> stack
fact(n => 0)
fact(n => 1)
fact(n => 2)
fact(n => 3)
```

We can examine the value of the innermost parameter `n` simply by typing its name. But to examine other dynamic occurrences of `n`, a position number must be used. In this case, the numbers correspond to their positions within the stack, the "outermost" (least recent) `n` having the largest position number. For example, at this breakpoint we can look at all invocations of `n`:

```
> n
0
> n[1]
0
> n[2]
1
> n[3]
2
> n[4]
3
```

Note that `n` and `n[1]` refer to the same object.

10.15 Examining Memory

The raw dump command (`raw`) enters a mode in which memory locations may be dumped in "raw format", as bytes, integers, and other primitive data formats. This mode should only be used as a last resort; it is usually much easier to look at memory using normal Ada variable references.

10.15.1 Command Forms

When the debugger goes into raw dump mode, the prompt changes to "`r:`". There are two command forms accepted in raw dump mode: a memory address to dump, or a quit command to return to the main debugger command level.

A memory dump command in raw dump mode has the form:

hex–address/format

This command form prints the value at the specified *address* using the specified *format.*

An address includes a segment and an offset separated by a colon. If there is no colon, the input is assumed to be an offset, and the previously specified segment is used.

To leave raw mode, type Control–Z (^Z) and press ENTER. The prompt changes back to a right angle bracket (">"), and all normal debugger commands may then be given.

10.15.2 Formatting Information

Memory locations may be examined by specifying a hex address, optionally followed by formatting information. The formatting information specifies how a data value is to be dumped (as a byte, an integer, or some other primitive data type). If the formatting information is missing, the last format specified is used. If the address is missing, then the previous address is incremented by an amount appropriate to the size of the data format, and that next location is dumped. This allows you to step through memory without having to re–specify an address or a format; just a carriage return may be given to request a dump of the next location.

Formatting information consists of a slash "/" followed by a *size* spec, a *type* spec, or both. The *size* spec specifies the size of the object to be printed in bytes. Valid *size* values are 1, 2 or 4. The *type* spec is a letter specifying as what data type the data is to be printed. The type formats are:

Type Specifier	Format
`b`	Print as a binary integer, *size* bytes long
`c`	Print as a character, *size* ignored
`d`	Print as a decimal integer, *size* bytes long
`o`	Print as an octal integer, *size* bytes long
`r`	Print as a real, *size* ignored
`p`	Print as a pointer (hex address), *size* ignored
`x`	Print as a hex integer, *size* bytes long

10.16 Debugger Command Reference

The debugger calls are listed below with their parameters. Items in `typewriter` font are literals to be typed as part of the command; *italicized* items indicate information that you must supply. A parameter listed in square brackets [] is optional; the square brackets are not to be typed as part of the command. An example command follows.

`breakp`(*subp* [, *line*])

Here, **`breakp`** is a command to be typed literally; *subp* can be a subprogram name and line is a specific line number.

Note: If a debugger command collides with a user–defined subprogram name, the debugger command must be qualified with the prefix **`debug.`**, as in:

```
debug.breakp(myproc)
```

If you have named a local variable, using the same name as a debugger command, then typing the command displays the value of the variable. This means that the debugger command does not execute. To execute the debugger command instead of displaying the variable value, type:

debug . [*command_name*]

See Section 10.13 for more information.

Some commands that do not require parameters have been assigned to function keys. For a complete list, see section 10.9.4.

---

at – execute a command at a specified location

Format **at**(*subp* [, *line*]) *command*

Description Setting an **at** breakpoint causes the specified *command* to be executed whenever the specified *line* in subprogram *subp* is encountered. This command has a slightly unusual form; the *command* to be executed appears after the parameter list. Any legal debugger command line is accepted for *command*. The subprogram *subp* must be visible when the **at** command is issued (see Section 10.13). If *line* is specified, then the *command* is inserted at the specified source statement line number in *subp*; if *line* is not specified, then the *command* is inserted at the beginning of *subp*; if *line* is –1 then the *command* is inserted at the end of the subprogram.

A command inserted with **at** is removed with the **unbreakp** command.

Note: All compilation unit initializations are completed before the first debugger prompt, so it is not possible to insert commands with **at** in the initialization sections of these units.

For more information, see section 10.7.2.

---

breakp – set a breakpoint

Format **breakp**(*subp* [, *line*])

Description The **breakp** command sets a breakpoint at the subprogram *subp*. The subprogram must be visible when the breakpoint is set (see Section 10.13).

If *line* is specified, then the breakpoint is set at the specified source statement line number; if *line* is not specified, then a breakpoint is set at the beginning of the subprogram; if *line* is –1 then a breakpoint is set at the end of the subprogram.

A breakpoint is cleared with the **unbreakp** command.

Note: All compilation unit initializations are completed before the first debugger prompt, so it is not possible to set breakpoints in the initialization sections of these units.

---

c – continue from breakpoint

Format **c**

Description The **c** command continues execution from a breakpoint. For more information, see section 10.6.2.

dump (subprogram) – print values of variables

Format **dump**(*subp*)

Description The **dump** command prints ("dumps") the values of all local variables and subprograms in the subprogram *subp*.

The **dump** command may also be used to dump packages (see next section).

For more information, see section 10.5.3.

dump (package) – list subprograms and print values of variables

Format **dump**(*package*)

Description The **dump** command, when applied to a package, dumps the values of all global variables and lists all subprograms in the specified *package*.

Variables and subprograms occurring in both the specification and body sections of a package are listed.

For more information, see section 10.5.3.

dumpall – dump current program status

Format **dumpall**

Description The **dumpall** command dumps the values of all local variables and subprograms in active subprograms, starting from the most recently activated subprogram (see Section 10.13) and ending with the main subprogram.

Following the local information about the main subprogram, the values of all global variables in all units are dumped, starting from the main program, and proceeding through the "highest" units (those at the top of the unit hierarchy), and going through the "lowest" units (those units that would be initialized first).

For more information, see section 10.5.3.

excbreak – select exception breakpoint handling

Format **excbreak**(*Boolean*)

Description The **excbreak** command specifies what action should be taken by the debugger when an exception is raised.

If the *Boolean* parameter is given as true, a debugger breakpoint occurs whenever an exception is raised by the program.

If the *Boolean* parameter is given as false, the effect is undone; normal program–defined exception handling is performed.

---

execute – execute a system command

Format `execute`(*command*)

Description The specified Ada string is executed by the operating system. The string need not be constant; a variable, even a slice or a function call may be used. For more information, see section 10.10.3.

---

find – locate a source code fragment

Format `find`[(*pattern*)]

Description The `find` command is useful when printing a large subprogram. After a source code fragment is listed by the `print` command, a subsequent `find` command will search the current subprogram for the specified `string` *pattern* parameter and resume printing there.

The *pattern* parameter may be any expression of type `string`.

If the *pattern* parameter is omitted, the `string` pattern given in the last `find` command is used.

---

go – run to temporary breakpoints

Format `go`[(*line* [, *alternate–line*])]

Description The `go` command continues execution from a breakpoint until the specified *line* is reached. An *alternate–line* may be specified. Two lines may be specified so that, for example, lines in the **then** and **else** parts of an **if** statement may be specified.

If no lines are specified, the next line in the current subprogram is executed. Note that this behavior is slightly different from the single–step command `ss`; a `go` command skips over a subprogram call, while an `ss` command steps into the call.

A `go` command never executes past the end of the current subprogram, unless the current breakpoint is at the end of a subprogram already.

For more information, see section 10.6.2.

---

help – print debugger help information

Format `help`

Description The `help` command prints a help message. The message lists the available debugger commands.

---

list – list execution controls and monitors

Format `list`

Description The `list` command lists breakpoints, tracepoints, and watches. All previously set breakpoints, tracepoints, and watches are listed.

---

more – continue source code listing

Format `more`

Description The `more` command prints ten more lines of the source code for the subprogram specified with the last `print`, `where`, or `find` command.

---

print – list source code fragment

Format `print(` *subp* [, *line*])

Description The `print` command reads the source file containing the subprogram *subp* and prints ten lines of the source code for *subp*.

If no *line* number is specified, then printing starts at line 1 of *subp*.

See also `more` command.

---

quit – exit debugger

Format `quit`

Description The `quit` command terminates the current debugging session and returns to the DOS command interpreter.

---

raw – enter raw memory dump mode

Format `raw`

Description The `raw` command enters raw address dump mode.

Refer to Section 10.15 for more information about this debugger command.

---

set_input – redirect debugger command input

Format `set_input(`*file*`)`

Description The `set_input` command redirects input from the specified file. The file name parameter may be any expression of type `string`.

Commands are read and executed from *file* until the end of the file is reached, whereupon the debugger accepts commands from the terminal again.

If a program being debugged does lots of terminal input, the debugger's I/O may interfere with the program's use of the screen. This command may be used to redirect debugger input to be from a file, or from another device:

```
set_input ("AUX:")
```

---

set_output – redirect debugger output

Format `set_output(`*file*`)`

Description The `set_output` command redirects debugger output to the specified *file*. The *file* parameter may be any expression of type `string`.

Output can be redirected back to the terminal by passing the empty string to `set_output`.

---

settaskname – rename task

Format `settaskname(` *task* [, *new–name*])

Description The `settaskname` command sets the "print name" of the specified *task* to be the specified string *new–name*.

If no *new–name* parameter is specified, the current name is printed.

---

silence – inhibit starting and ending breakpoints

Format `silence`

Description The `silence` command deletes the implicit breakpoints at the start and end of the program.

This is used most effectively in the debugger startup file (see Section 10.10.1).

---

source – reset default program source path

Format `source(`*path*`)`

Description The `source` command finds source files in the directory specified as the *path* parameter.

The *path* parameter may be any expression of type `string`. The specified directory *path* is used whenever a `print` command is given. The debugger always searches the current directory first.

Path is a string value conforming to file system conventions for directory specifications.

Example
```
source("c:\src")
```

SS – single–step

Format `ss`

Description The `ss` command executes a single program step. The code corresponding to a single source statement line is executed, following which control is returned to the debugger. If a call is executed, the breakpoint occurs at the start of the called subprogram.

Note that the `go` command can be used in place of `ss` to "skip over" a call.

For more information, see section 10.6.2.

---

stack – print stack trace

Format `stack[(`*levels*`)]`

Description The `stack` command prints a "stack trace". Information is displayed about the state of the call stack, which includes which subprograms have been called and with what parameter values.

Subprograms are listed in reverse order of activation, with the most recently activated subprogram being listed first, in the manner of the execution stack.

Levels indicates how many subprogram calls to display. If it is missing then the entire call stack is listed.

taskbreak – select task switch activity

Format `taskbreak(` *Boolean* `)`

Description The `taskbreak` command specifies what action should be taken by the debugger whenever a task context switch occurs.

If the *Boolean* state parameter is given as true, then a debugger breakpoint occurs at each task context switch. If the *Boolean* state parameter is given as false, then the effect is undone.

For more information, see section 10.12.4.

trace – trace subprogram execution

Format `trace(` *subp* `)`

Description The `trace` command sets a tracepoint in subprogram *subp*. A tracepoint may be cancelled with `untrace`. For more information, see section 10.7.1.

traceall – trace all subprograms

Format `traceall`

Description The `traceall` traces all subprograms. This effect may be cancelled with `untraceall`, but individual tracepoints may not be cancelled with `untraceall`. For more information, see section 10.7.1.

unbreakp – cancel breakpoint

Format `unbreakp(` *subp* [, *line*])

Description The `unbreakp` command deletes a previously set breakpoint at the subprogram *subp*. The subprogram must be visible when the breakpoint is deleted (see Section 10.13).

If no source statement line number *line* is specified, then the breakpoint at the start of *subp* is deleted; if *line* is specified, then the breakpoint at that source line number is removed.

untrace – cancel subprogram trace

Format `untrace(` *subp* `)`

Description The `untrace` command deletes a previously set tracepoint in subprogram *subp*. The `untrace` command does not affect the `traceall` command.

---

untraceall – cancel `traceall`

Format `untraceall`

Description The `untraceall` command cancels the effect of the `traceall` command. The `untraceall` command does not affect tracepoints set with the `trace` command.

---

unwatch – cancel variable watch

Format `unwatch(` *var* `)`

Description The `unwatch` command deletes a previously set debugger watch over the variable *var*.

---

watch – monitor the state of a variable

Format `watch(` *var* `)`

Description The `watch` command monitors the value of variable *var*. When the value of *var* changes, a message is printed showing the original value of the variable and its current value, and a breakpoint occurs.

A `watch` is cancelled with the `unwatch` command. For more information, see 10.6.3.

---

where – locate current source statement

Format `where`

Description The `where` command prints the line where execution is currently suspended. Five lines of "context" source code are printed before and after the current line.

The name of the currently executing task (if applicable) is also printed.

Chapter 11 Extended Mode Programs

This chapter documents the Extended Mode Meridian Ada compiler. This feature is not available in all versions of the compiler. Some of the discussion may be of interest to those with the standard DOS version of Meridian Ada. This chapter discusses how the Extended Mode Meridian Ada compiler can be used to create programs that run in Extended Mode.

Meridian Ada programs can run in Real Mode or Extended Mode. The primary difference between the two modes is in the amount of memory that a program can occupy and use. Extended Mode programs can be much larger than Real Mode programs.

Important

Extended Mode Meridian Ada is available separately from the standard DOS version of Meridian Ada. There are additional run–time licensing issues to consider for programs created with Extended Mode Meridian Ada that are intended for further distribution. These licensing issues are discussed in the *Licensing, Registration and Support* section of the Meridian Ada documentation package.

11.1 Extended Mode vs. Real Mode

The vast majority of programs that run under DOS are smaller than 640K and operate in Real Mode. Real Mode programs run on any DOS system, barring any peculiar system dependencies. Meridian Ada produces Real Mode programs by default. Computers based on 8086 or 8088 processors, such as the IBM PC/XT and IBM PS/2–30, run only in Real Mode.

Extended Mode Meridian Ada programs can take advantage of up to 16 megabytes of extended memory. Computers based on 80286 or 80386 processors, such as the IBM PC/AT, IBM PS/2–60, and IBM PS/2–80, can accommodate extended memory and may run either in Real Mode or in Extended Mode.

Regardless of the kind of processor used, programs running under DOS generally operate only in Real Mode and cannot use more than 640K of memory without special support. The Extended Mode version of Meridian Ada provides the necessary special support to use any extended memory installed in a system.

11.2 Benefits of Extended Mode

Extended Mode is also known as Protected Mode because the operating system is largely protected from applications running in that mode. In Real Mode, the operating system and application programs co–exist in the same address space, leaving the operating system vulnerable to any problems in application programs that may crash the system. In Extended Mode, there is a separation of the address spaces that protects the operating system against most common problems that may arise in application programs, such as attempts to write into code space or unassigned memory. Such problems are usually caught before they can crash the system and, in Extended Mode programs, raise the `program_error` exception.

This additional protection can help during the debugging phase of a program, especially when used with the Meridian Ada Debugger.

In addition to a better degree of protection from erroneous program behavior, Extended Mode operation also provides access to extended memory of up to 16 megabytes. The Extended Mode compiler uses extended memory to permit much larger programs to be compiled.

11.3 Expanded vs. Extended Memory

It is important to make the distinction between expanded and extended memory resources. Expanded memory using the Lotus–Intel–Microsoft (LIM, also called EMS) protocol permits areas of memory to be switched among each other within the 1MB address space of Real Mode. A Meridian Ada program may be built to make use of LIM expanded memory, provided that the program performs the low–level operations to switch the memory banks and the necessary bookkeeping to keep track of what objects are in which memory bank. There is no special run–time support presently in Meridian Ada that automatically takes advantage of LIM expanded memory resources. The compiler itself does not use LIM expanded memory.

Extended memory uses a conventional segment addressing scheme that permits up to 16 megabytes of memory to be addressed at one time. Extended memory is directly supported by the addressing modes of 80286 and 80386 processors, and can be used by Extended Mode programs. The Extended Mode compiler itself takes advantage of any installed extended memory to compile much larger programs than are possible with the Real Mode compiler.

11.4 Using Extended Mode

The following chart takes you through the basic steps for using Extended Mode. More detailed explanations of the steps are located in the sections that follow.

1. Create an executable program using the **ada** command. This program can be created in either Real or Extended Mode.

   ```
   ada prg_name.ada
   ```

   You can use command options just as you normally would with any real mode program.

2. Link the program using **bamp** with the command option **-x**.

   ```
   bamp -x prg_name
   ```

   The **-x** option creates an executable with a **.exp** extension.

3. Reconfigure the compiler to run in Extended Mode using the **adaext** command. (This step can be performed at any point in the process but it must be performed before you use **ramp** to run the program. See section 11.4.2 for more information.)

   ```
   adaext
   ```

4. Use the **ramp** command to run the program.

   ```
   ramp prg_name
   ```

11.4.1 Creating Extended Mode Programs

To create an Extended Mode Program use the **-x** option with **bamp** as shown in the following example:

```
bamp -x program_name
```

The Extended Mode configuration of Meridian Ada can be used to create either Real Mode or Extended Mode programs that run under DOS. The determination of whether a program is to run in Real Mode or Extended Mode is made at link time, using the **bamp -x** option to select Extended Mode. If the **-x** option is not given, programs are created to run in Real Mode.

If the program being compiled is to run in Extended Mode, it may be desirable to use the **ada -fS** option to produce 80286 instructions.

11.4.2 Increasing Compiler Capacity Using Adaext

If a program is very large, or consists of a large number of compilation units, it is possible for the compiler to run out of memory. With the Extended Mode compiler, it is possible to run the compiler itself in extended mode, freeing more memory to use during compilation.

By default, the compiler runs in real mode, with the normal memory limitations. The command, **adaext**, reconfigures Meridian Ada so that the compiler, optimizer and linker take advantage of extended memory. Extended Mode remains in effect until it is undone by the **adareal** command which returns the compiler to Real Mode.

The mode in which the compiler is run has no relationship to the mode in which the resulting program is run. For example, a program can be created to run in extended mode using a compiler running in real mode, or a real mode program can be created by running the compiler in extended mode. The **adaext** and **adareal** commands control the mode in which the compiler runs, and the **-x** option to **bamp** specifies the mode in which the created program should run.

11.4.3 Running Extended Mode Programs

There are two kinds of Extended Mode programs: *bound* and *unbound*. All Extended Mode programs are initially unbound, and to run them, the **ramp** command is used.

Bound programs may be created with the separately available **bind** command. The **bind** program combines the program with all the special run–time code necessary for your program to operate in Extended Mode into a single DOS **.exe** file that runs just like any other DOS program without the need for the **ramp** command. Contact Meridian Software Systems to obtain the **bind** program and an Extended Mode run–time license.

The **bamp -x** option produces an unbound Extended Mode program file with the extension **.exp**. An unbound Extended Mode program is started by using the **ramp** program, as in this example:

```
ramp extp
```

This example loads and runs an extended mode program file named **extp.exp**. It is not necessary to supply the **.exp** extension. Command line arguments or I/O redirections may be given on the same command line, as in this example:

```
ramp extp -o file.out <  data.txt
```

The only difference in invoking an unbound Extended Mode Meridian Ada DOS program vs. a normal DOS program is that the Extended Mode invocation is prefixed with the **ramp** command.

When the **ramp** program is used to invoke a **.exp** program, the **ramp** program first loads the Extended Mode support software (OS/x86), then loads and runs the Extended Mode program.

The **ramp** command requires that the **adaext** command has been run. This is only an issue if the compiler is being run in real mode to generate programs running in extended mode. For the more common case of running the compiler and the created program in extended mode, the **adaext** command should be in effect already. The reason for running the real mode compiler is that it is a little bit faster than the extended mode compiler. This cost should be weighed against the cost of having to run **adaext** before **ramp**ing the program and **adareal** after **ramp**ing.

11.5 System Requirements for Extended Mode

Extended Mode Meridian Ada programs can run on the vast majority of 80286 and 80386–based computers that run DOS. Systems that do not have the correct type of processor to use Extended Mode include the IBM PC/XT (8088 CPU) and the older model IBM PS/2–30 (8086 CPU). The Real Mode Meridian Ada compiler does run on the older IBM PS/2–30 (8086 CPU) and the IBM PC/XT (8088 CPU).

Although Extended Mode Meridian Ada is expected to run on most 80286 and 80386–based computers with DOS, the following systems do not accommodate Extended Mode Meridian Ada; however, Real Mode program are accommodated:

- Televideo
- ARC 286 Turbo
- Orchid PC Turbo

11.6 Distributing Extended Mode Programs

Extended Mode Meridian Ada programs operate in conjunction with a DOS–extender facility called OS/x86, which is licensed for use with Meridian Ada by ERGO Computing Solutions. The terms of this license are given in the *Licensing, Registration, and Support* section of the Meridian Ada compiler documentation package. An Extended Mode Meridian AdaVantage program intended for further distribution requires a separate run–time licensing arrangement. Contact Meridian Software Systems, Inc. for details.

11.7 DOS Compatibility in Extended Mode

Extended Mode Meridian Ada programs are DOS compatible; most Meridian Ada programs do not have to be modified to run in Extended Mode, even those using facilities in the DOS Environment Library and theMeridian Ada Utility Library. A few DOS facilities available in the DOS Environment Library are not meaningful or are not supported in Extended Mode; these are documented separately in the *OS/x86 Developers Reference Manual.* Contact Meridian Software Systems, Inc. to obtain this document.

11.8 Effects of Extended Mode on Memory Organization

The memory organization of Meridian Ada programs is documented in Chapter 13. The same organization applies both to Real Mode and to Extended Mode programs. Extended Mode Meridian Ada programs have access to more code space and heap space, depending on the amount of extended memory installed in the system. Extended Mode programs are also allocated more main program stack space by default, but the upper limit on stack space is the same for both Real Mode and Extended Mode programs.

The meaning of an address is slightly different in Extended Mode. Refer to section 14.9 for a discussion of machine addresses.

Interfaces between Extended Mode and Real Mode programs are possible; these are discussed in the *OS/x86 Developers Reference Manual.* Contact Meridian Software Systems, Inc. to obtain this document.

11.9 Effects of Extended Mode on Run Time Performance

The Extended Mode Meridian Ada compiler and Meridian Ada programs running in Extended Mode tend to run somewhat more slowly than their Real Mode counterparts. This is primarily because of the way DOS operations are supported under Extended Mode. To use DOS, which resides in the base 640K memory area and runs only in Real Mode, programs are automatically switched between Extended Mode and Real Mode. All disk file operations, for example, use DOS. Because there is no direct support (in the 80286 processor) for switching from Extended Mode to Real mode, the switch requires the processor to be reset, which also restarts the BIOS. This tends to slow things down.

Chapter 12 Groupware

Groupware is a collection of library management features that maintains the integrity of libraries in multi–user environments such as UNIX and Local Area Networks (LANs).

In a large scale programming project it is likely that more than one programmer might be accessing a given library at the same time. If the library is being updated by one programmer then all other programmers' access should be blocked until the update is complete. The Groupware components accomplish this goal on certain networks and multi–user systems.

Note: This feature is not available in all versions of the compiler.

12.1 System Requirements

Groupware features are included for DOS Extended Mode Meridian Ada, which is provided with the Meridian Ada product. The Meridian Ada product is shipped to you with both an Extended Mode and real mode compiler, however multi–user integrity is guaranteed only when all users on the LAN use the Extended Mode compiler. This minimizes the memory use of the real mode compiler, which can be constrained in the 640K memory space of the real mode DOS.

Groupware relies on file sharing services provided in DOS 3.1, and should work with any LAN product which supports those services. Groupware has been tested on Novell's Netware v2.15, Sun's PC–NFS v3.01, and on Invisible Software's NET/30 v1.50.

Using Groupware with PC–NFS and a Sun server requires PC–NFS 3.0.1 and SunOS 4.0.1 or later. The host file system must be mounted in sharing mode. In order for the host file system to be mounted in sharing mode you must use the **`/mustshare`** switch with the **`net use`** command or the **`ms`** option for the **`nfsconf`** "mount" menu. This may only work on Sun file systems; other types of UNIX may not support PC–NFS properly. You must have the **`rpc.lockd`** daemon running on each server machine. If a Groupware process seems to be hung for no apparent reason, check to see whether **`rpc.lockd`** is still active throughout the system. If **`rpc.lockd`** daemon is not, then restart it on the appropriate machine.

12.2 Using Groupware

The local library is accessed in an exclusive mode that prevents other Meridian Ada tools from reading or writing to it. Non–local libraries are accessed in a sharing mode that prevents other Meridian Ada tools from writing to them but allows them to be read. The **`lslib`** and **`bamp`** commands also access a local library in the sharing mode since they do not update the library.

When you try to access a library that is already in use, Groupware issues the message:

```
... waiting for access to file
```

At this point you can either wait or cancel your process using Control–C (^C). Keep in mind that while you are waiting, you could be preventing someone else from accessing some of the libraries you are using. It is also possible that while you are waiting for access to a library, the programmer with control of that library might be attempting to access a library that you are using. This form of circular deadlock should also be considered when linking libraries and planning the library structure for a programming project. Above all, do not attempt to have two or more users compiling the same module into the same library at the same time.

12.3 Library Creation and Maintenance Tracking

To provide helpful tracking for project management purposes a user name and group name is automatically associated with each new library unit compiled into the library, and on each subsequent recompilation. In addition, the **modlib** command can be used to provide user specific information in the library header.

You can communicate a user name and group name to Groupware by setting the DOS environment variables **user** and **group** to the desired values on your workstation. These values are limited by the DOS command buffer size to less than 118 characters.

The person that initially creates a Meridian Ada library will have their user name and group name (if available) recorded in both the **Library created** and the **Library updated** fields.

The person that initially adds an entry to the library, by compiling a source file, will have their user name and group name (if available) recorded in both the **Entered on** and **Changed on** fields. Subsequent updates will replace only the **Changed on** field for the units in that file.

If user and group information is available in the library it is displayed in conjunction with the related field. For example, if user *Joe* in the *Project_X* group updated a unit in a library, a line similar to the following would be seen in the library header listing:

```
Library updated on Mon Aug 28 1989 16:27:55 PDT by <Joe, Project_X>.
```

If either a user's name or their group name were unavailable at the time that the library was changed then the corresponding item in the display would be left out. If both were missing then the entire **by** clause would be omitted from the above listing line.

12.4 Managing Project Libraries

Although Meridian Ada tools can coordinate library access between themselves, when using non–Groupware programs, care must be taken to avoid situations where data consistency is at risk. For example, be careful not to edit a file or rename a library while it is being compiled.

A new command, **modlib**, is available to help manage a programming project's libraries. With it you can arbitrarily lock a library against update or insert additional documentation information in the library header. See section 19.12 for more information on using **modlib**.

The drive letter and path used for mapping to a server must match the corresponding drive and path used during the Meridian Ada installation or library creation process.

The documentation and internal library name strings will be limited by the DOS command buffer size to less than 118 characters.

12.5 System Considerations

Executable (**.exe**) files need to be marked as read–only with the DOS **attrib** command if they are to be shared.

NOTE: If the compiler is being shared it should remain either in real mode or Extended Mode to prevent user conflicts. If this is not the case, then any user that wants to switch modes will need network permission in the **paclib** directory to create, delete and rewrite the **avext** and **ada** files.

A message similar to the following may occur (very rarely) when using the Ada compiler:

```
Sharing Violation error reading drive F:
Abort, Retry, Ignore?
```

OR

```
Network Error: file in use during CREATE A FILE. File = F:simple.ils
Abort, Retry, Fail?
```

The best reply is **a** for **Abort**, since this message should not occur during proper operation of the compiler. If the message recurs with subsequent uses of the compiler and you are unable to discern its cause, try using **i** for **Ignore** once before replying **Abort**. Sometimes the **Ignore** reply allows the error to percolate up to a process that can issue a more intelligible message. If you are sure that you know why the message is occurring and still want to continue compiling then **Retry** can be used. In general, it is not recommended that you use **Retry** or **Ignore** because results can be unpredictable for any reply other than **Abort**.

Chapter 13 Memory Organization

This chapter explains how programs use memory and offers several helpful hints for making the most of available memory.

13.1 Run–Time Memory Usage

The discussion in this section is not of concern to most programmers. It is primarily useful for those programmers who need a detailed understanding of how the 80x86 processor architecture affects the memory organization of a Meridian Ada program. Some of the information presented in this section tends to require a more advanced understanding of certain aspects of the 80x86 processor architecture.

There are several kinds of memory allocations that are presently bounded by the 80x86 segmented architecture. Meridian Ada currently supports one memory model, a medium–large model, which is used both in Real Mode and Extended Mode programs (see Chapter 11 for a discussion of Real and Extended Modes). This memory model permits large amounts of code and data, but places bounds on individual pieces of code and data. Simple work–arounds are described throughout this chapter for those programs that bump into architectural limits. Section 13.2 discusses options for working with and within memory restrictions.

13.1.1 Code Size

The size of a single compilation unit presently may not exceed 64K bytes of code. This is not usually a problem, because:

1. The total amount of code in a program is limited only by available memory; the limit is only on individual compilation units.
2. Ada is designed so that use of smaller, separately compiled units is very easy.

Each individual compilation unit is assigned its own code segment. The limit to the size of an 80x86 code segment is 64K bytes, hence the limit on the size of a compilation unit.

13.1.2 Global Data Size

The total amount of global data in an entire program (as opposed to a single compilation unit of a program) cannot exceed 64K bytes. This is not a serious problem because the 64K limit on global space does not apply to heap space. Programs can use access types and dynamic allocation instead of large global objects. Refer to section 13.1.5 for information about how to do this.

One 80x86 segment is assigned for the global data. Since the data segment register is not changed by the program code, and global data offsets are only 16 bits wide, global data references are very fast, and the code is more compact. The limit on the amount of global data is the result of this choice of model.

13.1.3 Stack Size

The maximum amount of stack space that may be allocated for a program is 64K bytes. This limitation must be considered for:

- The amount of space allocated for local variables. Note that dynamic allocation may be used to work around stack space problems for local variables (see section 13.3).

- The number of tasks activated in a program (see Chapter 6).
- The activation depths of subprograms (the number of subprogram calls active at the same time).

A Real Mode application program that does not use tasking has 20K bytes of stack space allocated for it by default. If tasking is used, then an additional 20K bytes (40K bytes total) is allocated by default. The amount of stack space allocated to a program may be increased or decreased at link time by using the **-M** and **-s** options to **bamp**, but the absolute maximum is 64K bytes.

An Extended Mode application program that does not use tasking has the maximum 64K stack size allocated to it. If tasking is used, then 20K bytes within the 64K stack segment are reserved for use by tasks. As with Real Mode programs, the amount of stack space allocated to a program may be increased or decreased at link time by using the **-M** and **-s** options to **bamp**, but the absolute maximum is 64K bytes.

For an individual task, the default maximum stack size is 1024 (1K) bytes. This may be changed at compile time. Refer to Chapter 6 for an explanation of how to do this.

The program stack is assigned to one 80x86 segment. Since the stack segment register is not changed by the program code, and stack offsets are only 16 bits wide, stack references are very fast, and the code is more compact. The limit on the size of the program stack is the result of this choice of stack model.

13.1.4 Individual Data Object Size

The size of a single data object may not exceed 64K bytes. This means specifically that very large arrays and records cannot be declared. Note that this is also not often a problem because linked lists and other dynamic data structures, which are not subject to global data or stack space limitations, are easily created by allocators (new) and destroyed with **unchecked_deallocation**.

Each data object must fit within 64K because indexes or offsets are sixteen bit quantities.

13.1.5 Heap Storage

The heap is an area of memory where certain dynamically instantiated objects are created, such as internal temporaries created by concatenation with "**&**" (see section 2.4.4) and objects instantiated with new. The amount of memory available for the heap is that which is left over after memory has been allocated for operating system overhead, the program stack, and program code. The heap grows as memory allocations are requested by a running program.

Heap storage for temporaries is reclaimed automatically on source block boundaries. Heap storage for objects allocated with new must be reclaimed by calls to instantiations of the generic procedure **unchecked_deallocation**.

See section 13.3 for an example of how **new** and **unchecked_deallocation** can be used.

As with any other object, a dynamically instantiated object must fit within 64K bytes, because indexes or offsets are sixteen bit quantities. An object of precisely 64K bytes may be instantiated.

The heap management scheme used by the Meridian Ada run–time depends on the DOS memory allocation and deallocation system calls.

13.1.6 Storage Collections

A storage collection is allocated from the heap when the scope of the associated access type definition is entered and deallocated when the scope is left.

13.2 Running Out of Memory

A program may run out of memory at the time it is loaded (i.e. it exceeds the capacity of DOS) or at some point when it is running. This section discusses what can be done if a very large application program exceeds

the capacity of DOS at load time. Some of the discussion may apply also to what can be done with **storage_error** exceptions. A more complete discussion of what to do when a **storage_error** exception is raised while the program is running is given in section 13.3. A discussion of what to do when the compiler runs out of memory is given in section 3.2.

If a large application program exceeds the capacity of DOS, some options are:

1. Scavenge memory by reducing or eliminating system facilities that compete with the application program. Section 3.2 offers some suggestions for doing this; many of the same system configurations that affect the compiler may also affect an application program.
2. Use the Meridian Ada Optimizer with your program. Chapter 7 discusses program optimization.
3. Split the program into several smaller cooperating programs. This may require use of the DOS **exec** function. The *DOS Environment Library* provides this capability.
4. Use Extended Mode. If more than 1MB of extended memory is installed in the system, using Extended Mode permits a program to take advantage of the extended memory resources. Chapter 11 discusses Extended Mode.

13.3 Storage_Error Exceptions

The **storage_error** exception may be raised if, for example, you declare a very large array inside a procedure. Objects declared inside procedures (local objects) are placed in the program stack, where they occupy space only temporarily until the procedure returns. The amount of stack space is limited by the 80x86 architecture. Entering a procedure in which a 30,000–element array is declared may "blow up" the stack, resulting in a **storage_error** exception.

To explain in more detail, the amount of stack space available at a particular moment varies according to the amount of stack space consumed by each subprogram activation. Space consumed by an activation includes the space for the parameters, return addresses, and local objects. A local array is going to consume space on the stack when the subprogram containing the array is called. That space is reclaimed when the subprogram returns, but if the subprogram is called recursively, or many levels of subprogram calls are involved, then the additional amount of stack space consumed by each call must be considered.

The maximum stack size is 64K bytes, but only 20K is allocated to the stack by default unless you specify a larger stack size with the **-M** option to **bamp**. For example:

```
bamp -M 32000 alloc_demo
```

For a task, a length clause should be used, possibly in conjunction with the **-s** option to **bamp**.

Another way to manage the stack space is to use allocators, as in this example:

```
with unchecked_deallocation;
procedure alloc_demo(n: natural) is
  type presto_dynamo is array(natural range <>) of integer;
  type presto_dynamo_p is access presto_dynamo;
  procedure dispose is
    new unchecked_deallocation(presto_dynamo, presto_dynamo_p);
  dynamo: presto_dynamo_p;
begin
  if n > 0 then
    dynamo := new presto_dynamo(0 .. n - 1);
    for i in dynamo'range loop
      dynamo(i) := i;
    end loop;
```

```
        dispose(dynamo);
    end if;
end alloc_demo;
```

This example subprogram dynamically allocates a one dimensional array of *n* elements, indexed from 0 to *n* – 1, with **new**. In a **for** loop, the array is filled with numbers. When that is done, the array is disposed of and the space for it is reclaimed.

More importantly, the array access object *dynamo* only takes up eight bytes on the stack. The array itself is created dynamically on the heap, leaving lots of room on the stack for more subprogram calls, parameters, and local objects. Note that except for the set–up code (the declarations and the one **new** statement) and the take–down code (the **dispose** call), there is little or no visible difference between using an allocated array and an ordinary array.

Chapter 14 Internal Data Representations

This chapter describes internal data representations used by Meridian Ada. Address clauses, representation specifications, and the effects of pragma **pack** are also discussed.

14.1 Discrete Types

This table shows the number of bits used by predefined discrete types:

DATA TYPE	UNPACKED	PACKED
boolean	8	1
byte_integer	8	8
character	8	7
integer	16	16
long_integer	32	32

Type **boolean** is represented using an eight–bit byte, unless it is present in a packed composite type, in which case it may be compressed to a single bit.

Type **character** is represented using an eight–bit byte. The representation of type **character** may be slightly modified in a packed record, but not in a packed array (effectively).

Types **byte_integer**, **long_integer**, and **integer** are 8, 16 and 32–bit twos–complement integers. The representations of the pre–defined integer types are unaffected by their presences in packed composite types. Programs that must rely on specific precisions of integers for portability should use range definitions:

```
type int8  is range -128..127;          -- 8-bit integer type
type uns8  is range 0..255;             -- 8-bit unsigned type
type int16 is range -32_768..32_767;    -- 16-bit integer type
type uns16 is range 0..65_535;          -- 16-bit unsigned type
type int32 is range -2_147_483_648..2_147_483_647;
                                        -- 32-bit integer type
```

An enumeration type occupies 8, 16, or 32 bits depending on the number of values declared for the enumeration.

In general, a discrete type not appearing in a packed composite type is represented in the smallest number of bits, rounded up to the nearest number of bytes, that accommodates the range of values in that type. No subtype is represented in a smaller number of bits than its base type. Consider this declaration:

```
subtype small_range is natural range 0..255;
```

The subtype **small_range** in this example is actually represented in 16 bits.

A discrete type appearing in a packed composite type is packed to the smallest number of bits that accommodates the range of values in that type. Consider these declarations:

```
subtype small_range is natural range 0..255;
type byte_array is array(1..64) of small_range;
  pragma pack(byte_array);
```

The type **byte_array** is represented in 64 bytes.

A discrete type with a dynamic bound is always represented in the same number of bits as the base type. An example follows:

```
n: constant integer := 7;
subtype count is positive range 1..n;
```

Type **count** in this example is represented in 16 bits whether or not it appears in a packed composite type.

The effects of packing are discussed further in sections 14.4, 14.5.4 and 14.6.2.

14.2 Real Types

This table shows the number of bits used by real types:

data type description	bits
fixed point, byte	8
fixed point, short	16
fixed point, long	32
floating point	64

The representations of real types are unaffected by their presences in packed composite types.

14.2.1 Floating Point Representations

Meridian Ada uses a 80x87 64–bit floating point format for all floating point objects. This format conforms to the IEEE 754 standard for floating point.

14.2.2 Fixed Point Representations

Fixed point objects are represented as machine–integer objects with imaginary decimal points. Whether a fixed point type is "byte", "short", or "long" depends on the **delta** and **range** values specified for the type.

Length clauses to specify fixed–point **'small** values are implemented. Just as an application note, this facility is useful where exact representations are needed. An example follows:

```
type money is delta 0.01 range -1_000_000.0 .. 1_000_000.0;
for money'small use 0.01;
```

The length clause for **money'small** allows exact (unrounded) computations in increments of 1/100. Note that this is perhaps a naive approach to real–world financial computations; take it as a simple example only. In the absence of the length clause, the Ada language requires that the default value of **money'small** be $1/2^7$, a more efficiently handled value, but a value that can lead to some roundoff errors.

14.3 Enumeration Representation Clauses

Enumeration representation clauses, which are described in section 13.3 of the LRM, are supported. An enumeration representation clause is permitted following the type declaration to which it applies, provided that the representation clause occurs within the same declarative part as the type declaration. Note that the type declaration and the accompanying representation clause cannot be split across a package specification and body.

Here is an example program that demonstrates the use of enumeration representation clauses:

```
with text_io;
with unchecked_conversion;

procedure stepper is

  type command is (
      none,
      forward,
      backward,
      stop,
      motor_on);

  for command use (
      none     => 16#00#, -- 0 decimal
      forward  => 16#10#, -- 16
      backward => 16#11#, -- 17
      stop     => 16#1e#, -- 30
      motor_on => 16#1f#  -- 31
    );

  function value is new unchecked_conversion(source => command,
                                             target => integer);

  function field(s: in string; width: in positive)
      return string is
    f  : string (1 .. width) := (others => ' ');
    si : positive;
    fi : positive;
  begin
    si := s'first;
    fi := f'first;

    while si <= s'last and then fi <= f'last loop
      f(fi) := s(si);
      fi := fi + 1;
      si := si + 1;
    end loop;

    return f;
  end field;

  use text_io;
begin
  for i in command loop
    put_line(field(command'image(i), width => 10) & "POS  => " &
             integer'image(command'pos(i)) & ", VALUE => " &
             integer'image(value(i)));
  end loop;
end stepper;
```

The output of this program is:

```
none     pos  =>  0, value =>  0
forward  pos  =>  1, value =>  16
backwardpos   =>  2, value =>  17
stop     pos  =>  3, value =>  30
motor_onpos   =>  4, value =>  31
```

Note that there is no language-defined facility in Ada for obtaining the underlying values of enumeration elements for which a representation clause has been specified. In the Meridian Ada implementation, **unchecked_conversion** between the enumeration type and an integer type may be used to obtain the values.

14.4 Pragma Pack

Pragma **pack** is implemented for composite types (records and arrays).

Pragma **pack** is permitted following the composite type declaration to which it applies, provided that the pragma occurs within the same declarative part as the composite type declaration, before any objects of the composite type are declared.

Note that the declarative part restriction means that the type declaration and accompanying pragma **pack** cannot be split across a package specification and body.

The effects of pragma **pack** are discussed further in sections 14.5.4 and 14.6.2.

14.5 Array Types

14.5.1 Constrained Array Objects

A constrained array object with static bounds occupies just the space required for its elements. For example:

```
a: string(1..10);
 --a'first =  1
 --a'last =  10
```

In this example, object a takes up just the ten bytes required to hold ten characters (see Figure 14.1).

Figure 14.1 Constrained Array with Static Bounds

14.5.2 Dynamic Array Objects

A dynamic array object (an array with variable bounds) is represented as a pointer to a memory area containing the actual array elements. The memory area for the array elements is created on the dynamic allocation heap. Separate temporary objects are created to hold the variable bounds of the array; these temporaries have the same duration as the dynamic subtype. Global "temporaries" with permanent duration are created for global arrays with dynamically computed bounds; local temporaries are created in the stack for local arrays with dynamically computed bounds. No storage overhead is incurred for a static bound; the compiler keeps track of such bounds. An example follows:

```
n: integer :=  7;
b: string(1..n);
    --b'first =  1
    --b'last =  n
```

In this example, object b takes up the space required for the pointer as well as the space required for the elements (see Figure 14.2). Also, a temporary is created to hold the upper bound of the array since a subsequent change to the value of n should not effect this dynamic array.

Figure 14.2 Dynamic Array

14.5.3 Unconstrained Array Objects

An unconstrained array object (also an array with variable bounds, typically a parameter to a subprogram) is represented as an array access object. An example follows:

```
procedure p(s: string);
...
p("abcde");
```

In this example, the parameter s is passed as an unconstrained array access object (see section 14.7).

14.5.4 Packed Arrays

The effect of pragma **pack** on an array type is to cause packing of discrete type array elements into 16–bit words.

element bit size	elements packed per word	bits left over in completely packed word
1	16	0
2	8	0
3	5	1
4	4	0
5	3	1
6, 7, 8	2*	2+2, 1+1, 0
9...16	1	7 to 0

* One element per byte; see below

Arrays whose elements are 6, 7, or 8 bits wide are packed one element per eight–bit byte to maximize efficiency of access. For example, an array of type **character** arranges the characters on eight–bit byte boundaries, with one unused bit per byte (hence the 1 + 1 notation in the table above). The alternative method, not

used, is to pack the characters on adjacent seven–bit boundaries, which would leave two unused bits at the end of each 16–bit word, and would involve more code to address each character.

For an example of the array packing method, consider these declarations:

```
subtype register_values is natural range 16#0#..16#1F#;

type register_file is array(1..6) of register_values;
  pragma pack(register_file);
```

The type **register_file** is represented in two 16–bit words, with three 5–bit elements per word. The ordering of packed array elements within a word is discussed in section 14.5.5.

Pragma **pack** has no effect when applied directly to an array of a composite element type. Instead, pragma **pack** should be applied to the type from which the array is built. An example follows:

```
subtype small_range is natural range 0..255;
    -- An object of this type normally consumes one word.

type comp is array(0..1) of small_range;
  pragma pack(comp);
    -- This packs type comp into one word instead of two.

type comp_array is array(1..10) of comp;
    -- An object of comp_array type will consume ten 16-bit words.
```

Here, to effect packing, pragma **pack** is applied to the composite type **comp**. If pragma **pack** were instead applied to **comp_array**, each element of **comp_array** would consume two 16–bit words instead of one.

14.5.5 Array Element Arrangements

In the absence of packing, elements of an array are arranged in memory so that the last (rightmost) indices vary faster than the first (leftmost) indices. In a single dimension, elements are arranged so that the low–order elements of the array occupy lower addresses and high–order elements of the array occupy higher addresses.

Consider these declarations:

```
type byte_array is array(1..4) of character; -- constrained array type
ba: byte_array := ('a', 'b', 'c', 'd');      -- constrained array object
```

Following initialization of the array object **ba**, its contents are shown in Figure 14.3.

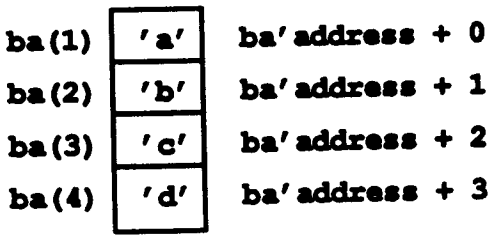


Figure 14.3 Array Element Arrangement for **ba**

Packed array elements are arranged with the first element occurring in the least–significant bits of the byte at the low–order address. This means, for example, that packed arrays of **character** are arranged exactly like unpacked arrays of **character**. For another example, consider this packed array type declaration:

```
type primary is (red, yellow, blue);
  --Each element of type primary can be represented in two bits.
```

```
type color_wheel is array(1..7)  of primary;
  pragma pack(color_wheel);
wheel: color_wheel := (blue, yellow, red, others => blue);
```

The object *wheel* is represented in a 16–bit word shown in Figure 14.4.

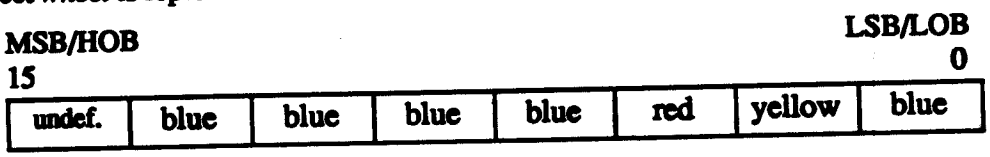


Figure 14.4 Packed Array Wheel

14.6 Record Types

The components of a record type are arranged in the order in which the components are declared.

In a discriminant record, the discriminants are treated as components of the record, and the discriminants occur as the first components of the record. For discriminant records whose discriminants may vary at run–time, an additional Boolean component appears at the start of the record. This component, the *is_constrained* flag, is used to implement the attribute `'constrained`. The *is_constrained* field occupies a byte for an unpacked record and a single bit for a packed record. The compiler omits the *is_constrained* component altogether if default values are not specified for the discriminants.

An unconstrained variant record object occupies as much space as is required to hold all the invariant components as well as the components of the largest possible variant of the record. When a specific discriminant value is given for a case selector, i.e. the record object is constrained, the size of the record reflects the selected variant.

14.6.1 Discriminant Array Components

A constrained array component in a record occupies as much space as a constrained array object of the same type. A discriminant array (an array whose bounds depend on discriminants) is represented as a pointer. An example follows:

```
type text(n: natural) is
  record
    val: string(1..n);
  end record;
x: text(5);  -- note: no default value for x.n
             -- x.val'first = 1
             -- x.val'last = x.n = 5
```

In this example, `x.val` is effectively a dynamic array, represented as a pointer to the elements of the array (see Figure 14.5).

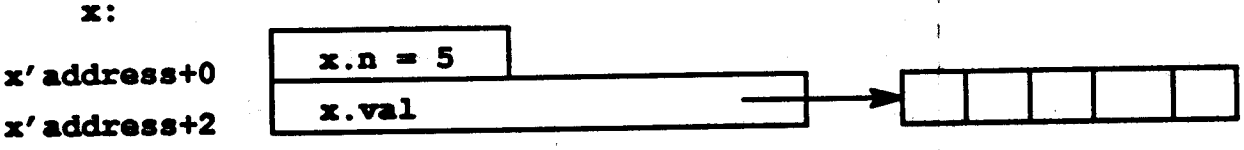


Figure 14.5 Record with Discriminant Array

14.6.2 Packed Records

The effect of pragma **pack** on a record type is to cause enumeration and discrete range components to be compressed so that they occupy the smallest number of bits appropriate to values of those types. An example follows:

```
type command_word_type is
  record
    rt_address : integer range 0..31;
    transmit   : boolean;
    sub_address: integer range 0..31;
    word_count : integer range 0..31;
  end record;

pragma pack(command_word_type);
```

Specifying pragma **pack** for type **command_word_type** is equivalent to specifying this record representation specification:

```
for command_word_type use
 record
    rt_address   at 0 * word range 00 .. 04;
    transmit     at 0 * word range 05 .. 05;
    sub_address  at 0 * word range 06 .. 10;
    word_count   at 0 * word range 11 .. 15;
end record;
```

The internal layout of an object of the packed type **command_word_type** is shown in Figure 14.6.

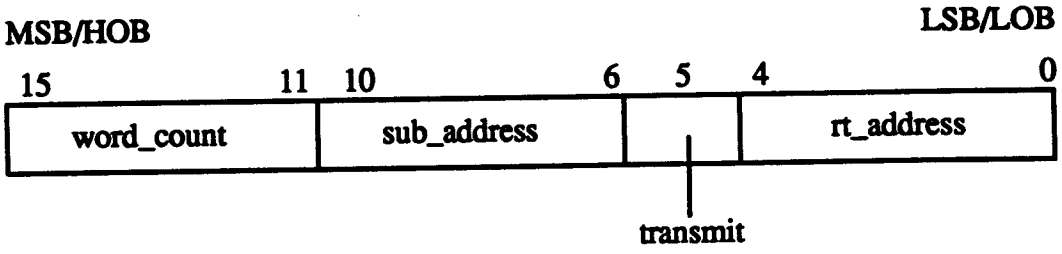


Figure 14.6 Packed Record

14.6.3 Record Representation Specifications

Record representation specifications, also called "rep specs", are implemented. A record rep spec is permitted following the record type declaration to which it applies, provided that the rep spec occurs within the same declarative part as the record type declaration, before any objects of the record type are declared. The stylistically preferred place of a record rep spec is immediately after the type declaration.

Note that the declarative part restriction means that the type declaration and accompanying rep spec cannot be split across a package specification and body.

Consider this record type declaration:

```
word: constant := 2;  -- Storage unit is byte, 2 bytes per word.
```

```
type command_word_type is
  record
    rt_address : integer range 0..31;
    transmit   : boolean;
    sub_address: integer range 0..31;
    word_count : integer range 0..31;
  end record;

for command_word_type use
  record
    rt_address   at 0 * word range 00 .. 04;
    transmit     at 0 * word range 05 .. 05;
    sub_address  at 0 * word range 06 .. 10;
    word_count   at 0 * word range 11 .. 15;
end record;
```

Given this rep spec, the internal layout of an object of type `command_word_type` is shown in Figure 14.6.

The current implementation–dependent restrictions on record rep specs are:

1. The storage unit offset (the `at` *static_simple_expression* part) is a word offset and must be even.
2. Bit positions (the `range` part) may be in the range 0..15, with 0 being the least significant bit of a component.
3. Components cannot straddle word boundaries.

If a compact representation for a record type is desired, but it is not necessary for the record to correspond to a particular externally determined layout, pragma `pack` should be used instead of a record representation specification.

14.6.4 Alignment Holes

To maintain alignments (see section 14.8), additional space can be allocated between components in a record. Consider these declarations:

```
type example is
  record
    a,b,c: character;
    i: integer;
  end record;

ex_rec: example;
```

In this example, the record occupies four bytes, not three. The internal layout of the record is shown in Figure 14.7.

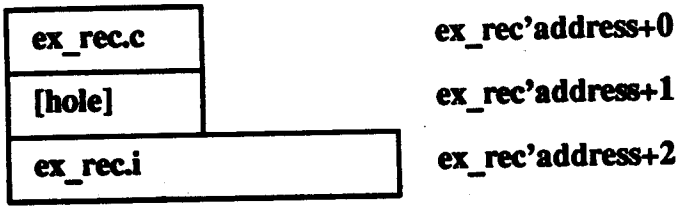


Figure 14.7 Record with Hole

If the declaration order of the components `c` and `i` is reversed in record type `example`, a hole is still allocated, as shown in Figure 14.8.

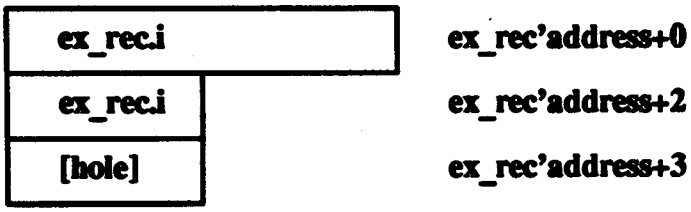


Figure 14.8 Rearranged Record with Hole

The hole at the end of the record is allocated because records can be elements of arrays or members of other records, and the alignment is then maintained for `ex_rec.i` in any data structure.

In general, the alignment of a record object reflects the alignment of the component with the strictest alignment requirement.

14.7 Access Types

An access object is potentially more than just a pointer. A pointer is a low–level object that references a memory location. Access objects for most types are represented simply as pointers to those objects. Access objects for arrays are usually just pointers as well, but access objects for unconstrained arrays are somewhat more complicated.

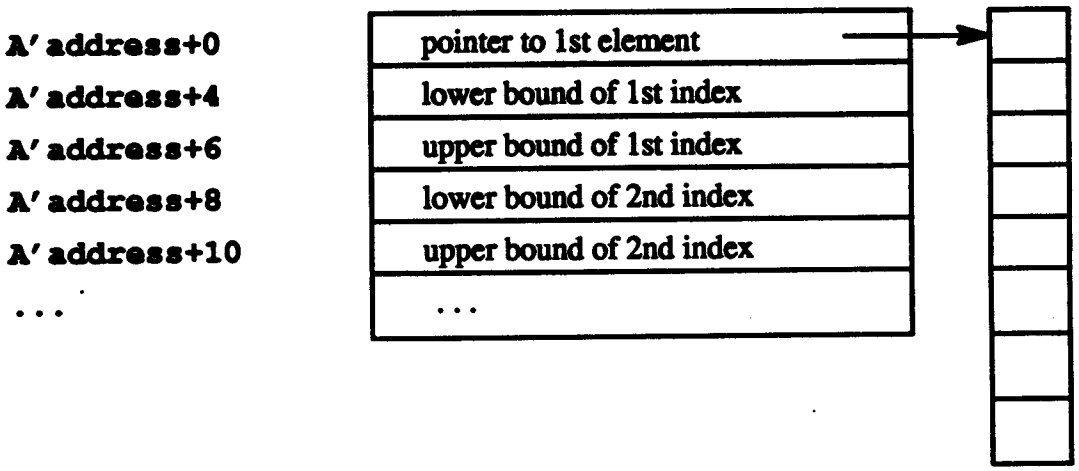


Figure 14.9 Array Access Type

An unconstrained array access object is represented internally as a structure containing:

- A pointer to the actual array data.
- The lower bound and upper bound of each index.

Figure 14.9 illustrates this structure.

An access object for a task type references a structure containing information about the task that is used by the tasking run–time support code. The internal layout of a task type is not documented by Meridian.

14.8 Alignments

For any address *a*, and a number *n* given in the table for the specified machine storage object, *a* mod *n* must be equal to zero to satisfy alignment requirements. The higher the value of *n*, the "more strictly aligned" a particular storage object is said to be.

Type	n
Boolean	1
character	1
integer	2

14.9 Address Clauses

An Address clause can be supplied for an object (whether constant or variable) or a task entry, but not for a subprogram, package, or task unit. An address clause is permitted following the declaration to which it applies, provided that the address clause occurs within the same declarative part as the item's declaration. The stylistically preferred place of an address clause is immediately after the declaration.

Note that the declarative part restriction means that the item's declaration and accompanying address clause cannot be split across a package specification and body.

14.9.1 Real Mode Object Address Clauses

In Real Mode, the address of an object is a 32–bit *simple_expression* of type **SYSTEM.ADDRESS** whose high–order 16 bits are the paragraph address and whose low–order 16 bits are the offset. A paragraph is 16 bytes, thus the 16–bit paragraph address **16#007F#** represents physical address **16#07F0#**; the paragraph address is shifted left four bits (simply catenate a hexadecimal zero on the right) to yield the physical address. The physical paragraph address resulting from shifting the high–order 16 bits of the address expression is then added to the offset in the low–order 16 bits of the address expression. For example, the address **16#007F_0008#** represents the absolute physical address **16#07F0#** + **16#0008#** yielding **16#0000_07F8#**.

Note that address expressions are signed 32–bit quantities, thus an address value larger than **16#7FFF_FFFF#** must be represented as a negative number. The negative number may be a universal integer expression that subtracts **16#1_0000_0000#** from the desired (unsigned) value. An example follows:

```
COLOR_GRAPHICS_SCREEN : SCREEN_ARRAY;

  -- where type SCREEN_ARRAY is some useful
  -- representation of the screen...

  for COLOR_GRAPHICS_SCREEN use at
    16#B000_8000# - 16#1_0000_0000#;

    -- Handy trick using universal integer
    -- computations to get 16#B000_8000#.
    -- This, by the way, represents physical
    -- address 16#000B_8000#, since the
    -- computation is 16#B_0000# + 16#8000#.
```

The Real Mode addressing scheme is discussed in the *Intel iAPX 86/88, 186/188 User's Manual/Programmer's Reference* (Reward Books, ISBN 0–8359–3035–1).

14.9.2 Extended Mode Object Address Clauses

In Extended Mode, the address of an object is a 32–bit *simple_expression* of type **SYSTEM.ADDRESS** whose high–order 16 bits are a *segment selector* and whose low–order 16–bits are an *offset* within a segment. An Extended Mode address expression does not directly represent a physical address. The segment selector is an index into an address table maintained by the Extended Mode support facilities.

The Extended Mode (Protected Mode) addressing scheme is discussed in the *Intel 80286 and 80287 Programmer's Reference Manual* (Intel, ISBN 1–55512–055–5).

On DOS systems that are running Extended Mode on the main processor (not on a co–processor card or an "accelerator" card), a segment selector of 16#00B0# in Extended Mode maps to the monochrome screen memory area; a segment selector of 16#00B8# maps to the color screen memory area. There are more general techniques for obtaining access to physical resources such as screen memory. Contact Meridian Software Systems for the *OS/x86 Developers Reference Manual*, which contains additional documentation on DOS system programming in Extended Mode; information on Extended Mode system programming is generally beyond the scope of this *User's Guide*.

14.9.3 Task Entry Address Clauses

For information on task entry address clauses, see sections F.5.4 and F.5.5.

Chapter 15 Pragma Interface

By using pragma **interface**, it is possible to make calls to subprograms written in 80x86 assembly language, Microsoft–C, or Meridian–Pascal from Meridian Ada programs. This section discusses pragma **interface**, machine code insertions, and calling conventions.

15.1 Formal Description

Pragma **interface** specifies that a subprogram is written in some other language and that the definition of that subprogram resides in a separate object module.

The form of pragma **interface** in Meridian Ada is:

pragma interface (*language*, *subprogram* [, "*link–name*"]) ;

where:

language This is the interface language, one of the names **assembly**, **builtin**, **microsoft_c** or **internal**. The names **builtin** and **internal** are reserved for use by Meridian compiler maintainers in run–time support packages.

subprogram This is the name of a subprogram to which the pragma **interface** applies.

link–name This is an optional string literal specifying the name of the non–Ada subprogram corresponding to the Ada subprogram named in the second parameter. If *link–name* is omitted, then *link–name* defaults to the value of *subprogram*. Depending on the language specified, some automatic modifications may be made to the *link–name* to produce the actual object code symbol name that is generated whenever references are made to the corresponding Ada subprogram. The object code symbol generated for *link–name* is always translated to upper case. Although the Meridian object linker is case–sensitive, it is a rare object module that contains mixed–case symbols; at present, all Meridian 80x86 object modules use upper case only.

The limit to the number of characters in the *link–name* depends on the interface language:

assembly	40 characters
builtin	38 characters
internal	38 characters
microsoft_c	39 characters

It is appropriate to use the optional *link–name* parameter to pragma **interface** only when the interface subprogram has a name that does not correspond at all to its Ada identifier or when the interface subprogram name cannot be given using rules for constructing Ada identifiers (e.g. the name contains a '$' character).

15.2 Calling Conventions

The discussion in this section about calling conventions applies to all pragma **interface** languages *except* for **microsoft_c** (see section 15.6).

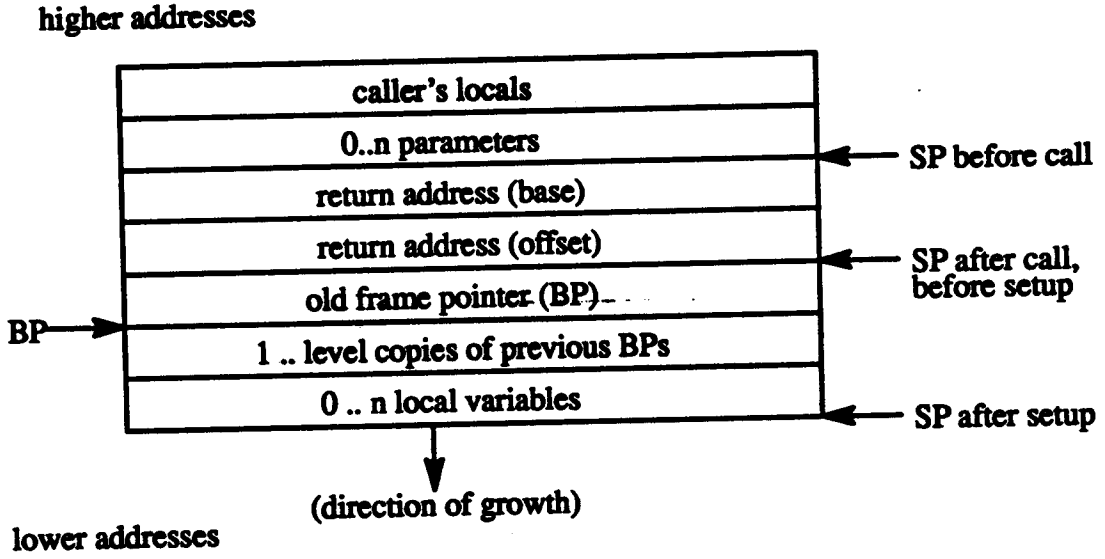
It is important to note that at present, no automatic data conversions are applied for any pragma **interface** subprogram parameters; if the data types of parameters are not directly compatible, then explicit conversions must be applied by you.

15.2.1 Stack Frames

Parameters are passed by pushing them onto the stack in reverse order of formal parameter declaration (right to left). The subprogram must know the number and types of the parameters in order to access the correct stack locations. All parameters are passed in the stack. Meridian Ada passes all non–scalar objects by reference, regardless of mode. Internal data representations are given in Chapter 14.

Caution must be exercised in passing parameters to pragma `interface` subprograms to maintain stack frames properly.

A representation of a stack frame is:



The stack frame for a subprogram consists of the actual parameters, the return address, a display, and the local variables. The display, which consists of copies of frame pointers from previous subprogram activations, is used to make references to local and non–local objects. An explanation of local objects is given in section 13.3.

A non–local object is a special case of a local object that is used for nested subprograms. When a subprogram makes a reference to a local object that is both non–global and not declared in the immediate subprogram scope, then the object is said to be non–local. The appropriate frame pointer from the in–stack display must be selected to make the necessary non–local reference.

Saving the previous frame pointers and allocating space for local variables is the responsibility of the called subprogram. It is also the responsibility of the pragma `interface` subprogram to deallocate the stack frame, including parameters, before returning. This can be done by specifying the appropriate number of bytes to deallocate with the terminating **RET** instruction.

Helpful Hint

The compiler is the definitive authority on how subprograms are called. If there are any questions about the calling conventions or internal data representations that are not answered in this documentation, then the answers may be determined by examining the assembly language source output from the compiler (see section 19.1). For example, a temporary stub subprogram can be written in Ada in place of the non–Ada subprogram. Both the calling subprogram and the stub can be compiled and the assembly language output examined to determine exactly how the calling conventions work in a particular case.

15.2.2 Register Usage

Only the registers SS, SP, DS, and BP must be preserved during the call. All other registers can be destroyed.

Register DS always points to the base of the global data segment, **$$DATA**.

Function return values are arranged as follows:

- 8–bit values are returned in register CL.
- 16–bit values are returned in register CX.
- 32–bit values are returned in registers BX:CX. The most significant word is in BX. the least significant word is in CX.
- 64–bit floating point values are returned in the 80x87 stack top register.

15.3 Code Model Compatibility

Meridian Ada presently uses a single code model that is described in Chapter 13. Non–Meridian language systems that do not use this model may require some additional **interface** code. Some specific aspects of the code model that affect compatibility are:

- Subprograms are called using 32–bit addresses. Using assembly language parlance, this means that pragma **interface** subprograms must be **FAR PROC**s.
- The compiler places all global data in a single segment named **$$DATA** of class **'DATA'**. If an Ada program is pre–linked with the **bamp -r** option, the data segment is renamed **_DATA**. If it is pre–linked with the **bamp -r -i** options, the data segment is renamed **DATA**. In any case, the data segment must not exceed 64K bytes. If a different data segment is used, then it is the responsibility of the pragma **interface** subprogram to save the DS register and restore it upon return.
- Meridian Ada code segments are of class **'CODE'**. The compiler places each compilation unit in its own logical segment, and switches the CS register for inter–segment calls.
- All segments must be aligned on paragraph boundaries.

15.4 Object Code Compatibility

The Meridian linker accepts a subset of the Microsoft object module format. This means in general that the Meridian linker can accept only Meridian object modules and Meridian libraries with a few rare exceptions, e.g. some version 4 Microsoft object modules. Where the Meridian linker cannot be used, the Microsoft linker, the DOS linker, and the Intel linker can accept Meridian object modules that have been pre–linked using the **bamp -r** or **bamp -r -i** options.

When pragma **interface** subprogram object modules are directly compatible with Meridian object modules, library augmentation with the **auglib** command can be used. An appropriate **auglib** command associates information with a Meridian Ada library unit so that the **bamp** command can obtain the names of external object modules that are to be linked with the rest of the program.

It is important to note that the Meridian linker is sensitive to case in symbol names (i.e. upper and lower case alphabetic characters are distinguished).

15.5 Interface to Assembly Language

An assembly language subprogram can be called by specifying the pragma **interface** language assembly. The *link–name* is used as the object code symbol as specified, with no modification.

Machine code insertions may be easier to deal with than assembly language pragma **interface** subprograms. Refer to section 15.9 for information about machine code insertions.

An example of an assembly language subprogram and the Meridian Ada **interface** code is:

```
; 8086 Assembly Language
Name    outport
.8087
DGROUP  GROUP   $$DATA
$$DATA  Segment Public  'DATA'
$$DATA  EndS
_outport Segment Public  'CODE'
Assume CS:_outport, DS:$$DATA, ES:NOTHING, SS:NOTHING
;----outport----
        Public  Outport
Outport Proc    Far
        push    bp          ;save old stack frame
        mov     bp,sp       ;set up stack frame
        mov     dx, 6[bp]   ;port
        mov     ax, 8[bp]   ;info
        out     dx,ax       ;move the bits
        pop     bp          ;restore the frame pointer
        ret     4           ;return and pop 4 bytes
                            ;of arguments
Outport EndP
_outport EndS
        END     ; of the program
```

```
-- Meridian Ada code:
package port is
  procedure put(port, item: integer);
    pragma interface(assembly, put, "outport");
end port;
```

```
with port;
procedure port_demo is
begin
  port.put(port => 16#60#, item => 16#41#);
end port_demo;
```

This example shows how to perform I/O port output using an assembly language routine.

For an assembler whose object code is directly compatible with the Meridian linker, the object module can be associated with the library unit **port** by using the **auglib** command:

```
auglib port outport.obj
bamp port_demo
```

This assumes that the **outport** object code is contained in a file named **outport.obj**. Note that **auglib** only adds the necessary information to the library entry for package **port**; **auglib** does not perform the actual linkage. The object file **outport.obj** is actually linked with the main program when **bamp** is invoked.

For other assemblers, the assembly code must be modified slightly and the object module must be linked separately with a pre–linked object module containing the Meridian Ada code.

For Microsoft assemblers, the data segment, **$$DATA**, must be renamed to **_DATA** and the program linked as follows:

```
bamp -r port_demo
link /nod /batch port_dem outport ;;;
```

For Intel assemblers, rename **$$DATA** to **DATA** and link as follows;

```
bamp -r -i port_demo
link /nod /batch port_dem outport ;;;
```

These are examples of how a non–Meridian linker might be used to link the object code for the **port_demo** program and the assembly language module **outport.obj**. Specific details vary with the linker used.

For further information on how to interface Meridian Ada programs with Intel assembly programs, see the **read.me** file in the **ada/inasm** directory.

15.6 Interface to Microsoft C

A subprogram written in Microsoft C can be called by specifying the language microsoft_c in pragma **interface**. This interface language changes not only the object symbol name, but also the calling conventions.

Object Code and Code Model Compatibilities

There is no standard Microsoft memory model that corresponds exactly to the memory model used by Meridian Ada. Microsoft C subprograms compiled using **/AL** (large model) can be used. This is an important consideration for linking with pre–compiled Microsoft C object code modules and object libraries; only libraries compiled with the correct code model can be used.

Meridian Ada programs with interfaces to Microsoft C must be partially linked using the **bamp -r** option and finally linked with the Microsoft linker.

The information in this section about code models, calling conventions, and linkage is valid for Microsoft C version 5.0.

Microsoft C Calling Conventions

The object code symbol generated for a microsoft_c interface subprogram prepends an underscore ("_") to the *link–name*.

The calling conventions are changed in the following ways for a microsoft_c subprogram.

- Function return values are given as follows:
  - 8–bit values are returned in register AL.
  - 16–bit values are returned in register AX.
  - 32–bit values are returned in registers AX and DX, with the most significant word in DX, the least significant word in AX.
  - Floating point values are discussed below.
- The caller (not the subprogram) is responsible for popping the actual parameters that the caller pushed onto the stack before the call.

These calling conventions are used automatically when pragma **interface** microsoft_c is used.

Data types are directly compatible for 8, 16 and 32–bit integer types as well as for access values (pointers). Enumeration types are directly compatible, except when passed as *reference* parameters (i.e. are passed by

address, as pointers). Microsoft C allocates an integer–size object for an enumeration type, while Meridian Ada allocates an object of a size appropriate to accommodate the range of values in the enumeration (8, 16, or 32 bits). Note that Ada out parameters are passed by reference.

For floating point values, Meridian Ada supports only type **double**, the 64–bit floating point type. Floating point actual parameters are directly compatible with formal C parameters of type **double**. Floating point function return values are presently not directly compatible. In Microsoft C functions, a **float** or **double** result is returned as a pointer to a floating point object. To call such a subprogram directly requires that the interface subprogram be specified as returning an access to a floating point object, and appropriate indirections made from there. Here is an example of how such an interface subprogram can be declared:

```
package fp_interface is
  type fp is access float;
  function f(x: float) return fp;
private
  pragma interface(microsoft_c, f);
    -- Microsoft c function declaration:
    -- double f(double x);
end fp_interface;
```

Note that because of the way Microsoft C returns floating point values as pointers to temporary storage, the caller should assign the result to a floating point object as soon as possible, as in this example:

```
x: float := fp_interface.f(3.7).all;
```

Data Transformations

No automatic data type transformations are made between Meridian Ada callers and Microsoft C callees. All internal data representations are those used by the Meridian Ada compiler.

Note in particular that Meridian Ada strings are not the same as C strings. Whereas a C string is simply passed as a pointer to a sequence of bytes in which the valid characters are terminated by an ASCII NUL character (value 0), an Meridian Ada string is passed as a pointer to an unconstrained array descriptor. A Meridian Ada subprogram that calls a Microsoft C subprogram expecting or returning C strings should be prepared to make the necessary conversions.

The source code of a package that provides a conversion facility between Meridian Ada strings and C strings and a demonstration program that uses the package follows. It is intended only to provide a guide to how such packages can be created.

File **cs.ads**:

```
with system;
package c_string_conversion is
  subtype c_string is system.address;
  procedure ada_to_c(astr: in  string; cstr: out string);
  procedure c_to_ada(
      cstr    : in c_string;
      str_len : out natural;
      astr    : out string);
end c_string_conversion;
```

File **cs.adb**:

```
with unchecked_conversion;
package body c_string_conversion is
```

```
procedure ada_to_c(astr: in  string; cstr: out string) is
begin
  cstr(1..astr'length) := astr;
  cstr(astr'length + 1) := ascii.nul;
end ada_to_c;

procedure c_to_ada(
    cstr    : in  c_string;
    str_len : out natural;
    astr    : out string) is
  len : integer;
  p   : c_string;
  ch  : character;

  type char_p is access character;

  function character_at is new
      unchecked_conversion(source => system.address,
                           target => char_p);

 begin -- c_to_ada
   len := 0;
   p := cstr;
   loop
     ch := character_at(p).all;
     exit when ch = ascii.nul;
     astr(astr'first + len) := ch;
     p := p + 1;
     len := len + 1;
   end loop;
   str_len := len;
 end c_to_ada;

end c_string_conversion;
```

Here is an example of how the conversion routines can be used:

File z.c:

```
/* C function to change each 'e' in a string into an 'x'.     */

void z(char *p)
{
  char *q;

  if (p)
    {
      for (q = p; *q; q++)
        if (*q == 'e') *q = 'x';
    }
}

int _acrtused;
/* This is defined just to resolve an MS C compiler-generated   */
/* external reference that is not used here but would otherwise */
/* require linking in the Microsoft-C run-time libraries.       */
```

File z_i.ada:

```
with system;

package z_i is
  procedure z(p: system.address);
private
  pragma interface(microsoft_c, z);
end z_i;
```

File demo_c_i.ada:

```
with c_string_conversion;
with z_i;
with text_io;
procedure demo_c_i is
  use c_string_conversion;

  c   : string(1..127);
  a   : string(1..127);
  len : natural;
begin
  ada_to_c(astr => "eschew Escher-esque entries", cstr => c);
  z_i.z(c(1)'address);
  c_to_ada(cstr    => c(1)'address,
           str_len => len,
           astr    => a);
  text_io.put_line(a(1..len));
end demo_c_i;
```

The example program can be compiled, linked, and run using the following commands, assuming that the specification and body of package c_string_conversion are split between two files named cs.ads and cs.adb, package z_i is in z_i.ada procedure demo_c_i is in demo_c_i.ada, and C function z is in z.c:

```
rem -- Compile the Ada units.

ada cs.ads cs.adb z_i.ada demo_c_i.ada

rem -- Partially link demo_c_i.

bamp -r demo_c_i

rem -- Compile the C unit with MS C using large model code (/AL).
rem -- Inhibit stack checking, which would drag in MS C run-time
rem -- libraries (/Gs).  Hold off linking (/c).

cl /AL /Gs /c z.c

rem -- Link everything with the MS linker.
rem -- Use no default libraries (/nod).
rem -- Ignore missing files (/batch).
rem -- Use all file name defaults (;;;).

link /nod /batch demo_c_i z ;;;

rem -- Run the linked program.

demo_c_i
```

The example, when compiled, linked and run, converts the Ada string "`eschew Escher-esque entries`" into a C string, calls a C function that changes all occurrences of the letter `e` to the letter `x`, changes the C string back into an Ada string, and produces this output:

```
xschxw Eschxr-xsqux xntrixs
```

Linking Meridian Ada Programs with Microsoft C

The `bamp -r` option produces a partially–linked file suitable for use with the Microsoft linker. From the example in the previous section, the `demo_c_i.obj` file that results from linking with the `-r` option contains all the necessary Ada run–time modules and all the program units, but not the object code for the C unit. The C unit is compiled separately and the entire program bound together using the Microsoft linker. Meridian Ada programs that interface to Microsoft C subprograms or libraries should always be linked in this manner. The normal `auglib` mechanism for associating non–Ada code modules with interface packages cannot be used for Microsoft C interfaces.

For information on how to interface Meridian Ada programs with Microsoft–C code that requires the use of the associated Microsoft–C run–time libraries, see the `read.me` file in the sub–directory, `msc`, under the Meridian Ada root directory (`c:\ada` in most installations). Contained in the `msc` directory are sample Ada and C program source files, batch files, and object modules. Although the interface techniques discussed above are adequate for interfaces to Microsoft C subprograms that do not use much of the Microsoft C run–time, subprograms that make use of Microsoft C's standard I/O, floating point types, and other facilities need to perform some additional set–up and take–down. Furthermore, the Microsoft C run–time libraries must be properly configured. The necessary techniques are discussed in the `read.me` file and explored in the sample program sources.

15.7 Interface to Meridian–Pascal

It is possible to make calls to Meridian–Pascal subprograms, but a matching Meridian–Pascal export interface pragma form must be used in the pertinent Meridian–Pascal unit:

```
(* Meridian-Pascal: *)
unit demo; interface
  pragma interface(c);
    procedure p;
  pragma interface(pascal);
implementation
  procedure p;
  begin
    ...
  end;
end.
```

```
-- Meridian Ada code:
package callit is
  procedure p;
    pragma interface(c, p);
end callit;
```

```
with callit;
procedure pascal_call_demo is
begin
  callit.p;
end pascal_call_demo;
```

In this example, the Meridian–Pascal procedure **demo.p** is made available to Meridian–Ada programs by matching the interface language (**c** in this case).

The reason that no interface **pascal** is available for Meridian–Ada is that in Meridian–Pascal, as in Meridian–Ada, compiler–assigned symbols do not look precisely like the programmer–assigned symbols, so Meridian–Ada programs making use of Meridian–Pascal symbols would most likely be that much more difficult to maintain. It is therefore easier to use **c** or **assembly** interface symbol correspondences.

Calling conventions between Meridian–Pascal and Meridian Ada are directly compatible only for scalar quantities and pointers. Meridian Ada passes all nonscalar objects by reference, regardless of mode. Parameter ordering is the same.

The **auglib** program must be used to associate the Meridian–Pascal subprogram with the calling program, as in this example:

```
auglib callit demo.obj
```

This assumes that the object code file containing the Meridian–Pascal function **demo** is named **demo.obj**. Note that **auglib** only adds the necessary information to the library entry for package **callit**; **auglib** does not perform the actual linkage. The object file **demo.obj** is actually linked with the main program when **bamp** is invoked.

15.8 Interfaces to Other Languages

Interfaces to other languages are possible provided that the symbol naming conventions and parameter passing conventions are the same or that interface subprograms are written to do any necessary conversions. Although only the interface languages mentioned in section 15.1 are supported, assembly interfaces can be used to perform any necessary conversions between calling conventions.

15.9 Machine Code Insertions

Machine code insertions, described in the LRM section 13.8, are supported in Meridian Ada. Several forms of machine code insertion can be used. Values must be static expressions. There are additional restrictions on machine code insertions, as described in the LRM:

- A compilation unit that contains machine code insertions must name package **machine_code** in a context (with) clause.
- The body of a procedure containing machine code insertions cannot contain declarations other than use, cannot contain statements other than code statements, and cannot contain exception handlers.

Refer to the LRM for more specific details.

The definition of package **machine_code** is:

```
package machine_code is
  type unsigned_byte is range 0..255;

  type inst1 is
    record
      b1: unsigned_byte;
    end record;

  type inst2 is
    record
      b1, b2: unsigned_byte;
    end record;
```

```
   type inst3 is
     record
       b1, b2, b3: unsigned_byte;
     end record;
   type instw is
     record
       b1: unsigned_byte;
       w: integer;
     end record;
   type immw is
     record
       w: integer;
     end record;
   type imml is
     record
       l: long_integer;
     end record;
 end machine_code;
```

The INST1, INST2 and INST3 records are used for creating one, two or three byte 80x86 instructions. The INSTW record is used to create a one–byte instruction with a two–byte immediate operand. The IMMW and IMML records are used for creating two–byte or four–byte operands.

```
with system;
with machine_code;
procedure urdbuf(file:   integer;
                 buf:    system.address;
                 nbytes: natural) is
  use machine_code;
begin
  -- This machine_code function issues a DOS read data
  -- from file request (INT 21, function code 16#3F#).
  inst1'(b1  =>  16#1E#);     -- PUSH  DS          ; save DS
  inst3'(16#C5#, 16#56#, buf'locoffset);
                              -- LDS   DX,[BP+08] ; DS:DX := buf
  inst3'(16#8B#, 16#5E#, file'locoffset);
                              -- MOV   BX,[BP+06] ; BX := file
  inst3'(16#8B#, 16#4E#, nbytes'locoffset);
                              -- MOV   CX,[BP+0C] ; CX := nbytes
  instw'(16#B8#, 16#3F00#);   -- MOV   AX,3F00    ; AH := 3F
  inst2'(16#CD#, 16#21#);     -- INT   21          ; issue DOS call
  inst1'(b1  =>  16#1F#)      -- POP   DS          ; restore DS
end;
```

The Meridian Ada attribute **'locoffset** is applied to a parameter and returns the stack offset of that parameter (the offset from the BP register). It allows machine code insertions to access parameters using less error–prone symbolic names.

Check your machine code insertions for correctness. This can be done by disassembling the final object code obtained from compilation of a subprogram containing machine code insertions. The DOS **debug** command can be used to do this.

15.10 Pragma Runtime_Names

Pragma **runtime_names** is intended primarily for use in conjunction with the Meridian Ada Run–Time Customization Library. It has the form:

```
pragma runtime_names(symbol-form);
```

where *symbol–form* is either **builtin** or **ada**.

The scope of this pragma is limited to the compilation unit in which it occurs (note that a package specification and its associated body are separate compilation units) or to the next occurrence of the pragma that uses a different *symbol–form*.

The effect of pragma **runtime_names** with *symbol–form* **builtin** is to modify the compiler–generated object code symbols generated for all identifiers falling within the scope of the pragma. The generated object code symbol name is the user defined identifier name prefixed with two underscores (__) (e.g. "example" becomes "__example").

Refer to the *Meridian Ada Run–Time Customization Library User's Guide* for examples of how to use this pragma.

Chapter 16 Standard Packages

16.1 Information About Standard Packages

The specification of package **system** is given in Appendix F in this document. Machine code insertions and package **machine_code** (LRM 13.8) are discussed in section 15.9.

The specifications of the remaining standard distribution packages are described in the LRM:

- package **calendar** (LRM 9.6)
- generic package **direct_io** (LRM 14.2.4)
- package **io_exceptions** (LRM 14.5)
- generic package **sequential_io** (LRM 14.2.1)
- package **standard** (LRM Appendix C)
- package **text_io** (LRM 14.3.10)
- generic function **unchecked_conversion** (LRM 13.10)
- generic procedure **unchecked_deallocation** (LRM 13.10)

Chapter 17 Additional Pre–defined I/O Packages

The full `text_io` package is designed to be very useful and flexible, but it can be difficult for novice Ada programmers to understand. For example, consider integer input and output. The standard `text_io` package does not directly define input–output for any particular integer type. Instead, a generic package, `integer_io`, must be instantiated for particular integer types. The standard Meridian Ada package includes several pre–defined packages designed to help overcome this potential problem:

`ada_io`	A simplified input–output package.
`fio`	An instantiation of `text_io.float_io` for type `float`.
`iio`	An instantiation of `text_io.integer_io` for type `integer`.

These packages are described in the following sections.

17.1 Package IIO

The Meridian pre–defined library package `iio` consists simply of an instantiation of `text_io.integer_io` for the standard type `integer`. This allows you to read and write integers without making your own instantiations of `integer_io`.

The source code for `iio` is simple:

```
with text_io;  use text_io;
package iio is new integer_io(integer);
    -- generic instantiated as library unit
```

Package `iio` can be used directly:

```
with iio;
procedure iexample is
 n: integer;
begin
 iio.get(n);
 iio.put(n);
end iexample;
```

Use of package `iio` not only makes integer I/O more convenient, but it also tends to be more space efficient, since multiple instantiations of the `integer_io` generic create distinct copies of the code.

17.2 Package FIO

The package `fio` instantiates `text_io.float_io` on the predefined type `float`:

```
with text_io;  use text_io;
package fio is new float_io(float);
 -- generic instantiated as library unit
```

This makes floating point I/O more convenient and space–efficient, just as `iio` does for integer I/O. An example follows:

```
with fio;
procedure fexample is
  r: float;
begin
  fio.get(r);
  fio.put(r);
end fexample;
```

17.3 Package Ada_IO

The package **ada_io** provides a simple interface to **text_io** for doing terminal input and output. **Ada_io** allows you to learn more quickly by hiding the more subtle aspects of the standard **text_io** package.

The package uses **text_io**, **iio**, and **fio** and consists simply of subprogram renaming declarations. For simple terminal input–output, a program only needs to specify **ada_io** in a with clause. The Ada compiler automatically uses the necessary declarations from **text_io** and the other packages.

The specification for **ada_io** follows:

```
with text_io;
with iio;
with fio;
package ada_io is

  procedure put(item: character) renames text_io.put;

  procedure put(
      item:  integer;
      width: text_io.field := 0;
      base:  text_io.number_base := 10
    ) renames iio.put;

  procedure put(
      item: float;
      fore: text_io.field := 0;
      aft:  text_io.field := 0;
      exp:  text_io.field := 0
    ) renames fio.put;

  procedure put(item: string) renames text_io.put;

  procedure put_line(item: string) renames text_io.put_line;

  procedure new_line(spacing: text_io.positive_count := 1)
    renames text_io.new_line;

  procedure get(item: out character) renames text_io.get;

  procedure get(
      item:  out integer;
      width: text_io.field := 0
    ) renames iio.get;

  procedure get(
      item:  out float;
      width: text_io.field := 0
    ) renames fio.get;
```

```
    procedure get_line(
        item: out string;
        last: out natural
      ) renames text_io.get_line;

    procedure skip_line(
        spacing: text_io.positive_count := 1
      ) renames text_io.skip_line;

    function end_of_line return boolean renames text_io.end_of_line;

  end ada_io;
```

By using Ada's renaming facility, **ada_io** simply changes the defaults for the **text_io** formatting parameters for integers and floats to values that have more intuitive effects (note that when a parameter to a subprogram has a default value, the parameter may be omitted in the call). Refer to the Ada reference manual for information about optional formatting parameters for **text_io** subprograms.

If the more advanced facilities of **text_io** are needed, then **ada_io** should most likely be excluded, although the packages **iio** and **fio** themselves can still be useful. Note that if both **ada_io** and **text_io** are made directly visible with use clauses, certain subprogram calls are ambiguous. For example, a call to **put** with a character parameter is ambiguous because the **put** in **ada_io** and the **put** in **text_io** are both visible. To get around this problem, the package can be specified explicitly, as in:

```
ada_io.put(ch)
```

An example of using **ada_io** follows:

```
with ada_io; use ada_io;
procedure for_demo is
begin
  for i in 1..9  loop
    for j in reverse 1..i loop
      put(j); -- This is ada_io.put.
    end loop;
    new_line; -- This is ada_io.new_line.
  end loop;
end for_demo;
```

When compiled, linked, and run, this program produces the output:

```
1
21
321
4321
54321
654321
7654321
87654321
987654321
```

Chapter 18 Using the Utility Library Packages

The *Meridian Ada Utility Library* is a set of Ada packages for use with the Meridian Ada compiler. The utilities provided with this package are:

Package `arg` Provides access to command line arguments.

Generic Package `array_object`
Provides ability to declare array objects larger than 64K.

Generic Package `array_type`
Provides ability to declare array types larger than 64K.

Package `bit_ops` Provides in–line bit–level operations.

Package `math_lib` Provides floating point transcendental functions.

Package `spio` Provides flushing for output text files.

Package `spy` Provides byte peek and poke operations.

Package `text_handler`
Provides a text handling facility as described in section 7.6 of the LRM.

The descriptions of the packages are organized alphabetically by package name. Each package is presented with its specification, a brief summary of its usage, and a discussion of the details of the package.

18.1 Package Arg

SPECIFICATION

```
package arg is
  function count return natural;
  function data(n: positive) return string;
end;
```

USAGE

Package `arg` provides data from the command line on which the program was invoked. This facility is similar to the `argc` and `argv` parameters provided to C programs. The command line is partitioned where one or more space characters (' ') occur. Each part, called a *command line argument*, is placed in a separate Ada string. The command line argument strings are ordered according to the position at which they occur on the command line, from left to right.

I/O redirections (">", "|") are discarded prior to program activation by the command line interpreter, thus such constructs do not appear in the command line data.

18.1.1 Function Count

Function `count` returns the number of command line arguments.

```
function count return natural;
```

The value returned by **count** is normally 1 or greater. The count includes the reserved index position 1. Valid command line arguments in PC–DOS therefore occur at positions 2 through **count**. If the count is 1, then there are no command line arguments.

For an example see function **data**.

18.1.2 Function Data

Function **data** returns the command line argument indexed by **n**.

```
function data(n: positive) return string;
```

Valid arguments occur at positions 2 through **count**. If **count** returns 1, then there are no command line arguments. A **constraint_error** is raised if **n** is less than 1 or greater than **count**.

An example of using package **arg** with function **data** follows:

```
-- This example accepts command lines of the form:
--
--      prog [-a] [-b] [arg] ...
--
-- where "prog" is the name of the program, "-a" and
-- "-b" are legal options, and "arg" is some optional argument.
--
with text_handler;          use text_handler;
with text_io;               use text_io;
with arg;
procedure scan_command_line_arguments is
  this: text(127);
  unrecognized_option: exception;
  illegal_argument: exception;
begin
  for i in 2..arg.count loop
    set(this, arg.data(i));
      -- convert string argument to text object.

    if not empty(this) then
      if value(this)(1) = '-' then
        if length(this) < 2 then
          raise illegal_argument;  -- argument was just '-' alone.
        end if;
        case value(this)(2) is
          when 'a' =>    put_line("-a recognized");
          when 'b' =>    put_line("-b recognized");
          when others => raise unrecognized_option;
                         -- argument wasn't "-a" or "-b".

        end case;
      else
        put_line("argument """ & value(this) & """ recognized");
          -- argument didn't start with '-'.
      end if;
    else
      raise illegal_argument;      -- argument was empty somehow.
    end if;
```

```
    end loop;
  exception
    when illegal_argument =>
      put_line("Illegal argument");
    when unrecognized_option =>
      put_line("Unrecognized option: '" & value(this) & "'");
  end scan_command_line_arguments;
```

18.1.3 Non–ASCII Characters

Note that it is possible (although difficult) to enter non–ASCII characters on a command line. Programs that do not require non–ASCII characters from the command line should "sanitize" command line data by clearing the eighth bits of characters in command line arguments (package `bit_ops` provides this capability). Otherwise, a program may generate `constraint_error` exceptions in places where non–ASCII characters in command line arguments are used and range checks are performed.

18.2 Generic Package Array_Object

SPECIFICATION

```
generic
  type index is (<>);
  type elem is private;
package array_object is
  procedure set(i: index; e: elem);
  function get(i: index) return elem;
end;
```

USAGE

Generic package `array_object` defines an individual array with subprograms `set` and `get` to manipulate elements of the array.

An example of an instantiation of this package is:

```
with array_object;
package a is new array_object(index => 1..10,
                              elem  => integer);
```

This instantiation is analogous to this array object definition:

```
a: array(1..10) of integer;
```

An example of instantiating and using this package is:

```
with array_object;
with ada_io;
procedure btest is
  type block is array(1..1000) of character;
  package a is new array_object(character, block);
  b: block;
  use ada_io;
begin
  for c in character loop
    b := (others => c);
    a.set(c, b);
  end loop;
```

```
    put_line("init done");
    for c in reverse character loop
      if a.get(c)(37) /= c then
        put_line("fail");
      end if;
    end loop;
    put_line("pass");
  end;
```

This program instantiates a package defining an array object, indexed by type `character`, of 1000–character blocks. The first for loop sets each 1000– character element to `NUL`, `SOH`, and on up to `'a'`, `'b'`, `'c'`, and so on. The second for loop checks the 37th character of each block to see whether it was properly initialized.

There are some special considerations for using package `array_object`:

- Since array elements must be copied when they are set or fetched, operations on large records may have significant overhead. It is best instead to define arrays of access types for efficiency.
- The size of an individual element must not exceed 16K bytes. The `storage_error` exception is raised if this limit is exceeded. Note that this is not a limit on the size of the entire array, although there must be enough memory in the system to accommodate the array.
- `Constraint_error` is properly raised for out of range indices.
- There is currently no way to de–allocate the object created by an instantiation of `array_object`.
- The size of an element of an array object created with `array_object` is rounded up to the next power of two so that array accesses are done more efficiently. Although this is not a factor for elements of discrete types or access types, it can apply to elements that are structures. For example, each array element whose type is a structure of size 17 bytes becomes 32 bytes in size. For this reason, it is more efficient in terms of storage to use an array of access types instead.

Note: Space for array objects created with generic package `array_object` is allocated from the heap, not from the stack or from global memory. This makes the best use of available memory.

18.3 Generic Package Array_Type

SPECIFICATION

```
generic
  type index is (<>);
  type elem is private;
package array_type is
  type desc(first, last: index) is limited private;
  procedure set(d: desc; i: index; e: elem);
  function get(d: desc; i: index) return elem;
private
  ...
end;
```

USAGE

Generic package **array_type** defines an array type with subprograms **set** and **get** to manipulate elements of objects of the array type. Whereas package **array_object** defines a single array object, package **array_type** permits several large arrays to be defined with similar characteristics.

The same considerations that apply to package **array_object**, discussed in the previous section, with respect to efficiency of element copying, deallocation, and storage efficiency, also apply to package **array_type**. As with **array_object**, space for array objects defined using generic package **array_type** is allocated from the heap, not from the stack or from global memory. This makes the best use of available memory.

An example of an instantiation of package **array_type** is:

```
with array_type;
package arr is new array_type(index => integer,
                              elem  => integer);
```

This instantiation is analogous to this unconstrained array type definition:

```
type arr is array(integer range <>) of integer;
```

An example of instantiating and using this package is:

```
with array_type;
with ada_io;
procedure mtest is
  type rec is
    record
      a, b, c: integer;
    end record;
  package arrtyp is new array_type(integer, rec);
  use arrtyp;
  r: rec := (0, 0, 0);
  a: desc(1, 20000);
  x: desc(0, 1000);
  use ada_io;
begin
  for i in a.first..a.last loop
    r.b := i;
    set(a, i, r);
  end loop;
  put_line("init done");
  for i in reverse a.first .. a.last loop
    if get(a, i).b /= i then
      put_line("fail");
    end if;
  end loop;
  put_line("pass");
end;
```

This example creates a package, **arrtyp**, that defines an unconstrained array type, indexed by **integer**, of records of type **rec**. An array of this type, **a**, constrained in the range **1..20000**, is then defined. A

second array, x, constrained in the range 0..1000, is also defined. Each component b of the elements of array a is initialized by a for loop to contain successive integer values beginning at 1. The same component is then tested by a second for loop to ensure that it contains the expected value.

18.4 Package Bit_Ops

SPECIFICATION

```
package bit_ops is
   --
   -- Operations on type byte_integer (eight bits):
   --

   function "and"(left, right: byte_integer) return byte_integer;
   function "or" (left, right: byte_integer) return byte_integer;
   function "xor"(left, right: byte_integer) return byte_integer;
   function "not"(left: byte_integer)        return byte_integer;
   function shl  (left: byte_integer; right: integer)
                                             return byte_integer;
   function shr  (left: byte_integer; right: integer)
                                             return byte_integer;

   --
   -- Operations on type integer (sixteen bits):
   --

   function "and"(left, right: integer)          return integer;
   function "or" (left, right: integer)          return integer;
   function "xor"(left, right: integer)          return integer;
   function "not"(left: integer)                 return integer;
   function shl  (left: integer; right: integer) return integer;
   function shr  (left: integer; right: integer) return integer;

   --
   -- Operations on type long_integer (thirty-two bits):
   --

   function "and"(left, right: long_integer) return long_integer;
   function "or" (left, right: long_integer) return long_integer;
   function "xor"(left, right: long_integer) return long_integer;
   function "not"(left: long_integer)        return long_integer;
   function shl  (left: long_integer; right: integer)
                                             return long_integer;
   function shr  (left: long_integer; right: integer)
                                             return long_integer;

end;
```

USAGE

Package `bit_ops` provides highly efficient bit–level operations. The functions in this package are specially optimized by theMeridian Ada compiler; they are translated directly into the equivalent machine–level operations rather than less efficient subprogram calls.

Note that all `bit_ops` functions are overloaded for `byte_integer`, `integer`, and `long_integer` types.

18.4.1 Function "And"

The function and returns the bitwise "and" of its operands.

```
function "and"(left, right: byte_integer) return byte_integer;
function "and"(left, right: integer     ) return integer;
function "and"(left, right: long_integer) return long_integer;
```

An example of using package bit_ops with function and follows:

```
with bit_ops;  use bit_ops;
procedure test_and is
  r: integer;
begin
  r := 0 and 0;  -- 0
  r := 0 and 1;  -- 0
  r := 1 and 0;  -- 0
  r := 1 and 1;  -- 1

  r := r and 16#7F#;       -- get low-order seven bits
end;
```

18.4.2 Function "Or"

The function or returns the bitwise "or" of its operands.

```
function "or"(left, right: byte_integer) return byte_integer;
function "or"(left, right: integer     ) return integer;
function "or"(left, right: long_integer) return long_integer;
```

An example of using package bit_ops with function or follows:

```
with bit_ops;  use bit_ops;
procedure test_or is
  r: integer;
begin
  r := 0 or 0;  -- 0
  r := 0 or 1;  -- 1
  r := 1 or 0;  -- 1
  r := 1 or 1;  -- 1

  r := r or 16#80#        -- turn on eighth bit
end;
```

18.4.3 Function "Xor"

The function xor returns the bitwise exclusive-or of its operands.

```
function "xor"(left, right: byte_integer) return byte_integer;
function "xor"(left, right: integer     ) return integer;
function "xor"(left, right: long_integer) return long_integer;
```

An example of using package bit_ops with function xor follows:

```
with bit_ops;  use bit_ops;
procedure test_xor is
  r: integer;
begin
  r := 0 xor 0;  -- 0
  r := 0 xor 1;  -- 1
  r := 1 xor 0;  -- 1
  r := 1 xor 1;  -- 0

  r := r xor 16#A5#;
end;
```

18.4.4 Function "Not"

The function **not** returns the bitwise negation of its operands.

```
function "not"(left: byte_integer) return byte_integer;
function "not"(left: integer     ) return integer;
function "not"(left: long_integer) return long_integer;
```

An example of using package **bit_ops** with function **not** follows:

```
with bit_ops;  use bit_ops;
procedure test_not is
  rb: byte_integer;
  rw: integer;
  rl: long_integer;
begin
  rb := not 0; -- 16#FF#
  rb := not 1; -- 16#FE#

  rw := not 0; -- 16#FFFF#
  rw := not 1; -- 16#FFFE#

  rl := not 0; -- 16#FFFF_FFFF#
  rl := not 1; -- 16#FFFF_FFFE#
end;
```

18.4.5 Function Shl

The function **shl** performs a logical left shift of the **left** operand by the number of bits specified in the **right** operand. Note that **right** is always of type **integer**.

```
function shl(left: byte_integer;  right: integer)
  return byte_integer;
function shl(left: integer;       right: integer)
  return integer;
function shl(left: long_integer;  right: integer)
  return long_integer;
```

An example of using package **bit_ops** with function **shl** follows:

```
with bit_ops;  use bit_ops;
procedure test_shl is
  rw: integer;
  rl: long_integer;
begin
  rw := shl(integer'(1), 1);                          -- 2
  rw := shl(integer'(1), integer'size - 2);           -- 16_384
  rw := shl(integer'(1), 0);                          -- 1

  rl := shl(long_integer'(1), 1);                     -- 2
  rl := shl(long_integer'(1), long_integer'size - 2);
                                                      -- 1_073_741_824
  rl := shl(long_integer'(1), 0);                     -- 1
end;
```

18.4.6 Function Shr

The function **shr** performs a logical shift right of the **left** operand by the number of bits specified in the **right** operand. Note that **right** is always of type **integer**.

```
function shr(left: byte_integer;  right: integer)
  return byte_integer;
function shr(left: integer;       right: integer)
  return integer;
function shr(left: long_integer;  right: integer)
  return long_integer;
```

An example of using package **bit_ops** with function **shr** follows:

```
with bit_ops;  use bit_ops;
procedure test_shr is
  rw : integer;
  rl : long_integer;
begin
  rw := shr(integer'(1), 1);                          -- 0
  rw := shr(integer'(16#4000#), integer'size - 2);    -- 1
  rw := shr(integer'(1), 0);                          -- 1

  rl := shr(long_integer'(1), 1);                     -- 0
  rl := shr(long_integer'(16#4000_0000#), long_integer'size - 2);
                                                      -- 1
  rl := shr(long_integer'(1), 0);                     -- 1
end;
```

18.5 Package Math_Lib

SPECIFICATION

```
package math_lib is
  pi: constant :=
    3.14159_26535_89793_23846_26433_83279_50288_41972;

  e: constant :=
    2.71828_18284_59045_23536_02874_71352_66249_77572;
```

```
    function sin (x: float) return float;
    function cos (x: float) return float;
    function exp (x: float) return float;
    function sqrt(x: float) return float;
    function ln  (x: float) return float;
    function atan(x: float) return float;
  end;
```

USAGE

The package **math_lib** provides some useful transcendental functions as well as the constants **pi** and **e** given to forty decimal places.

18.5.1 Function Sin

The function **sin** returns the sine of its parameter, **x**, given in radians.

```
function sin(x: float) return float;
```

An example of using package **math_lib** with function **sin** follows:

```
with math_lib;   use math_lib;
procedure test_sin is
  r: float;
begin
  r := sin(pi / 2.0);  -- 1.0
end;
```

18.5.2 Function Cos

The function **cos** returns the cosine of its parameter, **x**, given in radians.

```
function cos(x: float) return float;
```

An example of using package **math_lib** with function **cos** follows:

```
with math_lib;  use math_lib;
procedure test_cos is
  r: float;
begin
  r := cos(pi / 2);  -- 0.0
end;
```

18.5.3 Function Exp

The function **exp** returns **e** raised to the power of its parameter, **x**.

```
function exp(x: float) return float;
```

An example of using package **math_lib** with function **exp** follows:

```
with math_lib;  use math_lib;
procedure test_exp is
  r: float;
begin
  r := exp(0);  -- 1.0
end;
```

18.5.4 Function Sqrt

The function `sqrt` returns the square root of its parameter, `x`.

```
function sqrt(x: float) return float;
```

An example of using package `math_lib` with function `sqrt` follows:

```
with math_lib;  use math_lib;
procedure test_sqrt is
  r: float;
begin
  r := sqrt(2);  -- 1.41421 ...
end;
```

18.5.5 Function Ln

The function `ln` returns the logarithm to base `e` of its parameter, `x`.

```
function ln(x: float) return float;
```

An example of using package `math_lib` with function `ln` follows:

```
with math_lib;  use math_lib;
procedure test_ln is
  r: float;
begin
  r := ln(e);  -- 1.0
end;
```

18.5.6 Function Atan

The function `atan` returns the arctangent of its parameter, `x`, given in radians.

```
function atan(x: float) return float;
```

An example of using package `math_lib` with function `atan` follows:

```
with math_lib;  use math_lib;
procedure test_atan is
  r: float;
begin
  r := 4 * atan(1); -- pi
end;
```

18.6 Package Spio

SPECIFICATION

```
with text_io;
package spio is
  procedure flush(file: in text_io.file_type :=
                        text_io.current_output);
end;
```

USAGE

Package `spio` is intended for special manipulation of I/O run-time operations. Presently, `spio` provides one function, `flush`, which does flushing for output text files. This is useful for displaying individual strings on

terminals without calling **text_io.new_line**. Files associated with terminal devices are line buffered, so a **put** does not display anything until a newline is printed or the terminal output file (typically **current_output**) is flushed. **Spio.flush** is also useful in debugging to ensure that the maximum amount of text appears in an output file if a program is crashing mysteriously, and otherwise might not flush its last output to a file.

Note that it is not necessary to use **spio.flush** to do same–line prompting for terminal input; when a **get** operation occurs on a terminal device, all output associated with the terminal device is flushed automatically.

An example of using package **spio** follows:

```
-- This tests the spio.flush operation.
with text_io;
with spio;
procedure flushit is
  type string_p is access string;
  type wordlist is array(natural range <>) of string_p;
  word: constant wordlist := (
     new string'("This"),     new string'("is"),
     new string'("a"),        new string'("deliberately"),
     new string'("delayed"),  new string'("sequence"),
     new string'("of"),       new string'("words"),
     new string'("on"),       new string'("a"),
     new string'("single"),   new string'("line"),
     new string'("."),        new string'("."),
     new string'(".")
    );
begin
  text_io.put_line("This is an ordinary line of output");
  for i in word'range loop
    text_io.put(word(i).all);
    text_io.put(' ');
    delay 0.5;
    spio.flush;
  end loop;
  text_io.put_line("<done>");
end flushit;
```

This example shows how **spio.flush** can be used to display individual words on a line with a time delay between each word. Note that when **spio.flush** is called with no parameter, it defaults to **text_io.current_output**.

18.7 Package Spy

SPECIFICATION

```
with system;

package spy is
  subtype byte is natural range 0..255;

  function peek(addr: system.address) return byte;
  procedure poke(value: byte; addr: system.address);
end spy;
```

USAGE

Package **spy** is used to "spy" on memory locations or change their contents.

Note that type **spy.byte**, as a subtype of **natural**, is not an eight–bit quantity; however, for **peek** and **poke** operations, it is the case that only eight–bit byte locations in memory are examined or modified.

18.7.1 Function Peek

Function **peek** returns the value of the eight–bit byte at the absolute memory location specified as **addr**.

```
function peek(addr: system.address) return byte;
```

An example of using package **spy** with function **peek** follows:

```
-- This tests peek and poke from the spy package

with spy;
with text_io;
with system;
procedure pptest is
  q: character;
  b: spy.byte;
  v: constant spy.byte := 16#55#;

  package lio is new text_io.integer_io(system.address);
  package bio is new text_io.integer_io(spy.byte);
begin
  spy.poke(value => v, addr => q'address);
  text_io.put("Poked ");
  bio.put(item => v, width => 0, base => 16);
  text_io.put(" into location ");
  lio.put(item => q'address, width => 0, base => 16);
  text_io.new_line;

  b := spy.peek(addr => q'address);
  text_io.put("Peeked ");
  bio.put(item => b, width => 0, base => 16);
  text_io.put(" from location ");
  lio.put(item => q'address, width => 0, base => 16);
  text_io.new_line;

  if b = v then
    text_io.put_line("PASSED");
  else
    text_io.put_line("FAILED");
  end if;
end pptest;
```

18.7.2 Procedure Poke

Procedure **poke** sets the value of the byte at memory location **addr** to the **value** specified.

```
procedure poke(value: byte; addr: system.address);
```

See the example of function **peek** for an example of how **poke** is used.

18.8 Package Text_Handler

SPECIFICATION

```
package text_handler is
  maximum: constant := integer'last;
  subtype index is integer range 0 .. maximum;

  type text(maximum_length: index) is limited private;

  function length(t: text) return index;
  function value (t: text) return string;
  function empty (t: text) return Boolean;

  function to_text(s: string;    max: index) return text;
  function to_text(c: character; max: index) return text;
  function to_text(s: string               ) return text;
  function to_text(c: character            ) return text;

  function "&"(left: text;      right: text     ) return text;
  function "&"(left: text;      right: string   ) return text;
  function "&"(left: string;    right: text     ) return text;
  function "&"(left: text;      right: character) return text;
  function "&"(left: character; right: text     ) return text;

  function "=" (left: text; right: text) return boolean;
  function "<" (left: text; right: text) return boolean;
  function "<="(left: text; right: text) return boolean;
  function ">" (left: text; right: text) return boolean;
  function ">="(left: text; right: text) return boolean;

  procedure set(object: in out text; value: in text);
  procedure set(object: in out text; value: in string);
  procedure set(object: in out text; value: in character);

  procedure append(tail: in text;      to: in out text);
  procedure append(tail: in string;    to: in out text);
  procedure append(tail: in character; to: in out text);

  procedure amend(object: in out text;
                  by: in text;
                  position: in index);
  procedure amend(object: in out text;
                  by: in string;
                  position: in index);
  procedure amend(object: in out text;
                  by: in character;
                  position: in index);

  function locate(fragment: text;      within: text) return index;
  function locate(fragment: string;    within: text) return index;
  function locate(fragment: character; within: text) return index;
```

```
private
  type text(maximum_length: index) is
    record
      length: index := 0;
      value: string(1 .. maximum_length);
    end record;
end text_handler;
```

USAGE

The specification of package **text_handler** is taken from the LRM section 7.6. An object of type **text** is a variable–length string. Each **text** object has a maximum length, which is given as a discriminant when the object is declared. A **text** object also has a current dynamic length, a number between zero and the maximum length, representing the number of "valid" characters in the current text value. Any characters beyond the current length are undefined. The maximum possible length of a text object is **integer'last**.

Note that dealing directly with a **string** is not always quite as convenient as dealing with **text**, primarily because of the variable length aspect of **text**. A **string**, which is an unconstrained array, has a fixed length.

18.8.1 Function Length

Function **length** returns the current length of a **text** object.

```
function length(t: text) return index;
```

An example of using package **text_handler** with function **length** follows:

```
with text_handler;  use text_handler;
procedure test_length is
  t: text(10);
  l: index;
begin
  set(t, "rasp");
  l := length(t);  -- 4
end;
```

18.8.2 Function Value

Function **value** returns the **string** portion of a **text** object.

```
function value(t: text) return string;
```

An example of using package **text_handler** with function **value** follows:

```
with text_handler;  use text_handler;
procedure test_value is
  t: text(10);
  s: string(1..4);
begin
  set(t, "rasp");
  s(1..4) := value(t);    -- t must contain exactly 4 characters
end;
```

18.8.3 Function Empty

Function **empty** returns **true** if the current dynamic length of its text parameter is 0; **false** otherwise.

```
function empty(t: text) return Boolean;
```

An example of using package **text_handler** with function **empty** follows:

```
with text_handler;  use text_handler;
with text_io;       use text_io;
procedure test_empty is
  t: text(10);
begin
  set(t, "");  -- zero-length string
  if empty(t) then
    put_line("<empty>");
  else
    put_line("t = """ & value(t) & '"');
  end if;
end;
```

18.8.4 Function To_Text

The overloaded **to_text** functions convert **string** or **character** objects to **text** objects.

String to Text of Specified Size

The first version of **to_text** converts a **string** to **text**, returning a **text** value whose maximum size is **max**.

```
function to_text(s: string; max: index) return text;
```

Character to Text of Specified Size

The second version of **to_text** converts a **character** to **text**, returning a **text** value whose maximum size is **max**.

```
function to_text(c: character; max: index) return text;
```

String to Text of Minimum Size

The third version of **to_text** converts a **string** to **text**, returning a **text** value whose maximum size is **s'length**.

```
function to_text(s: string) return text;
```

Character to Text of Minimum Size

The fourth version of **to_text** converts a **character** to **text** of maximum length 1.

```
function to_text(c: character) return text;
```

18.8.5 Function &

The overloaded **"&"** functions concatenate **text** objects with other **text** objects, with strings, or with characters. A returned **text** result has a maximum length that is the sum of the current dynamic lengths of its operands.

```
function "&"(left: text;      right: text     ) return text;
function "&"(left: text;      right: string   ) return text;
function "&"(left: string;    right: text     ) return text;
function "&"(left: text;      right: character) return text;
function "&"(left: character; right: text     ) return text;
```

An example of using package `text_handler` with function `&` follows:

```
with text_handler;  use text_handler;
procedure test_concat is
  t: text(80);
begin
  set(t, "da");
  set(t, t & t);          -- "dada"
  set(t, t & "esque");    -- "dadaesque"
  set(t, '_' & t);        -- "_dadaesque"
end;
```

18.8.6 Relational Functions

The relational functions for `text` objects are essentially the same as the equivalent functions for strings, except that the current dynamic lengths are taken into consideration.

```
function "=" (left: text; right: text) return boolean;
function "<" (left: text; right: text) return boolean;
function "<="(left: text; right: text) return boolean;
function ">" (left: text; right: text) return boolean;
function ">="(left: text; right: text) return boolean;
```

Note that there is an implicit `"/="` function available as well.

An example of using package `text_handler` with relational functions follow.

```
with text_handler;  use text_handler;
with text_io;       use text_io;
procedure test_relop is
   t, u: text(10);
begin
   set(t, "there");
   set(u, "therapy");
   if t > u then
     put_line("PASSED-- t > u");
   else
     put_line("FAILED-- t <= u");
   end if;
end;
```

18.8.7 Procedure Set

The overloaded `set` procedures provide assignment operations for `text` objects.

```
procedure set(object: in out text; value: in text     );
procedure set(object: in out text; value: in string   );
procedure set(object: in out text; value: in character);
```

An example of using package `text_handler` with procedure `set` follows:

```
with text_handler;  use text_handler;
procedure test_set is
  t, u: text(10);
begin
  set(t, "zot");
  set(t, 'z');
  set(u, t);
end;
```

18.8.8 Procedure Append

The overloaded **append** procedures append **string**, **text** or **character** values to **text** objects.

```
procedure append(tail: in text;      to: in out text);
procedure append(tail: in string;    to: in out text);
procedure append(tail: in character; to: in out text);
```

An example of using package **text_handler** with procedure **append** follows:

```
with text_handler;  use text_handler;
procedure test_append is
  t, u: text(10);
begin
  set(u, "st");
  set(t, "pa");
  append(u,        to => t);  -- "past"
  append('e',      to => t);  -- "paste"
  append("urize", to => t);  -- "pasteurize"
end;
```

18.8.9 Procedure Amend

The overloaded **amend** procedures replace part of a **text** object starting at a particular index position. These provide a facility similar to string slice assignment. Note that the first **position** in a **text** object is 1.

```
procedure amend(object: in out text;
                by: in text;
                position: in index);
procedure amend(object: in out text;
                by: in string;
                position: in index);
procedure amend(object: in out text;
                by: in character;
                position: in index);
```

An example of using package **text_handler** with procedure **amend** follows:

```
with text_handler;  use text_handler;
procedure test_amend is
  t, u: text(10);
begin
  --       1234567
  set(t, "analogy");
  set(u, "yst")
  amend(t, by => u,    position => 5);  -- "analyst"
  amend(t, by => "ze", position => 6);  -- "analyze"
  amend(t, by => 's',  position => 6);  -- "analyse"
end;
```

18.8.10 Function Locate

The overloaded **locate** functions search for fragments (patterns) in **text** objects. Zero is returned if an attempt to locate a *fragment* in a **text** object fails.

```
function locate(fragment: text;      within: text) return index;
function locate(fragment: string;    within: text) return index;
function locate(fragment: character; within: text) return index;
```

An example of using package **text_handler** with function **locate** follows:

```
with text_handler;  use text_handler;
procedure test_locate is
  t, u: text(80);
  i: index;
begin
  --                1         2         3
  --      123456789012345678901234567890123456
  set(t, "Beware the Lurker at the Threshold.");
  set(u, "Cthulhu!")
  i := locate("Lurk", within => t);  -- 12
  i := locate('T',    within => t);  -- 26
  i := locate(u,      within => t);  -- 0
end;
```

18.8.11 Input–Output of Text

Here are examples of input–output routines for **text**:

```
with text_handler;       use text_handler;
with text_io;            use text_io;
procedure put(t: text) is
begin
  put_line(value(t));
end;
```

```
with text_handler;        use text_handler;
with text_io;             use text_io;
procedure get(t: in out text) is
  last: natural;
  s: string(1..t.maximum_length);
begin
  get_line(s, last);
  set(t, s(1..last));
end;
```

Part II Meridian Ada Command Reference

This part of the manual contains all Meridian Ada compiler commands in alphabetic order. Each command is listed along with its format, options if any, and examples where possible.

Chapter 19 Meridian Ada Command Details

The library management commands are described in this section, along with their invocation forms and their options. The commands are non–interactive; all necessary information is specified on the command line.

An option for a command is specified as a letter preceded by a dash character ("–"). Options are separated from each other on the command line by a space character (" "). An option may have more than one part: the sequence –*x*, where *x* is some letter, and an argument, separated from its preceding option letter *x* by a space.

Note: Upper and lower case option letters mean different things.

In the invocation forms shown, square brackets [ ] indicate optional information (not to be typed with literal brackets); *italicized* items indicate variable information; ellipses ("...") indicate that the preceding item or items may be repeated indefinitely. For example:

ada [*option* ...] *file*.**ada** ...

Here, **ada** is a command to be typed literally; *option* is some optional information that may be repeated; *file*.**ada** is file name information that may be repeated.

19.1 ada

19.1.1 Invocation

```
ada [option ...] file.ada ...
```

19.1.2 Description

The **ada** command invokes theMeridian Ada compiler. If the Extended ModeMeridian Ada compiler is installed, then the compiler runs in Extended Mode. There are no differences between invocations of the Extended Mode compiler and the standard Real Mode compiler. Extended Mode Operation is discussed in Chapter 11.

A program library must be created using **mklib** or **newlib** in advance of any compilation. The compiler aborts if it is unable to find a program library (either the default, **ada.lib**, in the current working directory or the library name specified with the **-L** option).

Note that the source file has the extension **.ada**. Just about any non–empty file extension is permitted. The ones not allowed include those used by the Meridian Ada compiling system for other purposes such as **.obj** for object module files. If an illegal extension is given, the error message "**missing or improper file name**" is displayed. Some other commonly used source file extensions are:

.ads for package specification source files

.adb for package body source files

.sub for subunit (separate) source files

The Meridian Ada compiler emits native 80x86 object code. Determination of whether a program is to run in Real Mode or Extended Mode is made at link time, using the **bamp -x** option to select Extended Mode. See section 19.7 for a discussion of the **bamp** command. Programs that are to run only in Extended Mode, or that are to run in Real Mode only on 80286 or 80386 processors, may be compiled with the **ada -fs** option to specify 80286–target code.

19.1.3 Options

-fD Generate debugging output. The **-fD** option causes the compiler to generate the appropriate code and data for operation with the Meridian Ada Debugger. For more information on using this option and using the Debugger, see Chapter 10.

-fe Annotate assembly language listing. The **-fe** option causes the compiler to annotate an assembly language output file. The output is supplemented by comments containing the Ada source statements corresponding to the assembly language code sections written by the code generator. To use this option, the **-S** option must also be specified, otherwise the annotated file is not emitted. Refer to section 19.1.9 for more information.

-fE Generate error log file. The **-fE** option causes the compiler to generate a log file containing all the error messages and warning messages produced during compilation. The error log file has the same name as the source file, with the extension **.err**. For example, the error log file for **simple.ada** is **simple.err**. The error log file is placed in the current working directory. In the absence of the **-fE** option, the error log information is sent to the standard output stream.

-fF Disable floating point checks. This option is used to inhibit checks for a math co–processor before sequences of math co–processor instructions, resulting in a slightly smaller and faster program. Use

of this option means that the resulting program requires, and you guarantee, the run–time presence of a math co–processor (either an 8087, 80287, or 80387). If a program containing floating point computations is compiled with the **-fF** option, it will behave unpredictably if run on a machine without a math co–processor installed; the machine may simply "freeze up" in this circumstance, requiring a reboot. Additional information about the floating point software is given in Chapter 9. Refer also to the **bamp -u** option, which causes the floating point software to be linked with a program.

-fL Generate exception location information. The **-fL** option causes location information (source file names and line numbers) to be maintained for internal checks. This information is useful for debugging in the event that an "Exception never handled" message appears when an exception propagates out of the main program (see section 3.5). This option causes the code to be somewhat larger. If **-fL** is not used, exceptions that propagate out of the main program will behave in the same way, but no location information will be printed with the "Exception never handled" message.

-fN Suppress numeric checking. The **-fN** option suppresses two kinds of numeric checks for the entire compilation:

1. **division_check**
2. **overflow_check**

These checks are described in section 11.7 of the LRM. Using **-fN** reduces the size of the code. Note that there is a related **ada** option, **-fs** to suppress all checks for a compilation. See also section 3.5.

The **-fN** option must be used in place of pragma **suppress** for the two numeric checks, because presently pragma **suppress** is not supported for **division_check** and **overflow_check**. Pragma **suppress** works for other checks, as described in section 2.4.2. In the absence of the **-fN** option, the numeric checks are always performed.

-fR Inhibit static initialization of variables. This option is intended for use in ROM–based embedded environments in conjunction with the Meridian Ada Run–Time Customization Library. The **-fR** option is applicable only in the presence of the **-fs** option, which suppresses certain runtime checks. Normally, the Ada compiler initializes constants or variables with static data when the following conditions all occur:

1. Checking is disabled with the **-fs** option.
2. The initializer expression is static (known at compile time).
3. The object is a global (in top–level package specification or body).

If the **-fR** option is specified, static initialization is suppressed for variables (but not for constants); assignments to each component of a variable are performed in the code. Note that this always happens in the absence of the **-fs** option.

-fs Suppress all checks. The **-fs** option suppresses all automatic checking, including numeric checking. This option is equivalent to using pragma **suppress** on all checks. This option reduces the size of the code, and is good for producing "production quality" code or for benchmarking the compiler. Note that there is a related **ada** option, **-fN** to suppress only certain kinds of numeric checks. See also sections 2.4.2 and 3.5.

-fS The **-fS** option causes the compiler to generate additional 80286 instructions not available on the 8086/8088. Programs compiled in this mode tend to be smaller than programs compiled using the normal 8086/8088 mode. Nothing special is required to link programs compiled using this mode; the usual **bamp** command is used. Note that by default, programs are created to run in Real Mode. If the **bamp -x** option is not used, the resulting program runs in Real Mode regardless of whether or not the program has been compiled with the **-fS** option. The **bamp -x** option must be used to produce an Extended Mode program.

WARNING:

Be certain that programs that run in Real Mode and that have been compiled for the 80286 (or, more specifically, the individual modules that have been compiled for the 80286) are not run on 8086 or 8088–based systems, such as the original IBM PC, IBM PC–junior, IBM PC/XT, IBM PS/2–30 or similar compatible systems. Programs compiled for the 80286 will run on either 80286 or 80386–based systems, such as the IBM PC/AT, IBM PS/2–60, IBM PS/2–80, Zenith Z–248, and similar compatible systems.

Programs that run in Extended Mode will not run on on 8086 or 8088 based systems.

-fU Inhibit library update. The **-fU** option inhibits library updates. This is of use in conjunction with the **-s** option. Certain restrictions apply to the use of this option; see section 19.1.9.

-fv Compile verbosely. The compiler prints the name of each subprogram, package, or generic as it is compiled. Information about the symbol table space remaining following compilation of the named entity is also printed in the form "[*n*K]".

-fw Suppress warning messages. With this option, the compiler does not print warning messages about ignored pragmas, exceptions that are certain to be raised at run–time, or other potential problems that the compiler is otherwise forbidden to deem as errors by the LRM.

-g The **-g** option instructs the compiler to run an additional optimization pass. The optimizer removes common sub–expressions, dead code and unnecessary jumps. It also does loop optimizations. This option is different from the **-g** option to **bamp**. The **-g** option to **ada** optimizes the specified unit when it is compiled; no inter–unit optimization is done. The **-g** option to **bamp** analyzes and optimizes the entire program at link time. Note: Even if **-g** is specified for the **ada** command, the **-K** option to **ada** must still be specified for the **-g** option to **bamp** to be effective.

-K Keep internal form file. This option is used in conjunction with the Optimizer (see Chapter 7 for more information). Without this option, the compiler deletes internal form files following code generation.

-l*modifiers*

Generate listing file. The **-l** option causes the compiler to create a listing. Optional modifiers can be given to affect the listing format. You can use none or any combination of the following modifiers:

c	continuous listing format
p	obey pragma **page** directives
s	use standard output
t	relevant text output only

The formats of and options for listings are discussed in section 19.1.7. The default listing file generated has the same name as the source file, with the extension **.lst**. For example, the default listing file produced for **simple.ada** has the name **simple.lst**. The listing file is placed in the current working directory. Note: **-l** also causes an error log file to be produced, as with the **-fE** option.

-L *library–name*

Default: **ada.lib**

Use alternate library. The **-L** option specifies an alternative name for the program library.

-s Produce assembly code. Causes the code generator to produce an assembly language source file and to halt further processing. This option must be used cautiously; refer to section 19.1.9 for more information.

Note: Options beginning with **-f** can be combined, as in "**-fsv**." This is equivalent to specifying the options separately, e.g. "**-fs -fv**." Options beginning with **-l** can be similarly combined or separated, as in "**-lcs**" or "**-lc -ls**" (see section 19.1.7).

19.1.4 Compiler Output Files

Files produced by compilations, other than listings and error logs, are:

.asm files	assembly language files (only when **-S** option used)
.atr files	interface description files
.int files	Meridian Internal Form Files
.g*nn* files	generic description files; *nn* is a two–digit number
.obj files	object code files
.sep files	subunit environment description files

Also produced are various intermediate files; these are usually deleted as a matter of course. You normally need not concern yourself with most of these output files with the exception of assembly language files.

Output files are placed either in the current working directory or in the auxiliary directory, depending on the configuration of the program library (as determined by **mklib** or **newlib**). The name of an auxiliary directory associated with a program library can be determined by using the **-h** option to the **lslib** command.

Supplementary files that may be produced by a compilation are:

.err files	error log files (only when **-fE** option used)
.lst files	listing files (only when **-l** option used)

These are discussed in section 19.1.7.

19.1.5 Non–Local Compilations

The compiler is able to compile files that reside in directories other than the current working directory. As always, a program library (typically **ada.lib**) must be present in the current working directory. All output files are placed in the current working directory or in the local auxiliary directory (**ada.aux**).

19.1.6 Compile–Time Error Messages

When syntactic or semantic errors are detected in the source code, the Meridian Ada compiler produces either error messages or warning messages. These messages are normally produced on the standard output stream. If the **-fE** option is given, these messages are written, instead, to an error log file. The error log file has the same name as the source file, with the extension **.err**.

When error messages are printed, processing does not proceed beyond the first pass. No object code file is produced. Warning messages do not prevent further processing. Other passes (e.g. the code generator) may print error messages as well, but these are almost certain to be error messages related to problems internal to the compiler itself, and should be reported to your distributor.

Error messages have the form:

"*filename*", *nn*: *English explanation of error* [**LRM** *l.m.n/p*]

where *filename* is the name of the program source file in which the error was detected, *nn* is the specific line number in the source file where the error occurred, followed by an explanation in English of the error, and, when appropriate, a reference to the LRM. The LRM reference gives the chapter (*l*), section (*m*), subsection (*n*), and paragraph number (*p*).

An example follows.

```
"rt.ada", 245: record component redeclared [LRM 3.7/3]
```

Depending on the severity of the error, the Meridian Ada compiler may attempt to recover and continue compiling the source code, or may terminate compilation immediately.

Warning messages have the form:

```
"filename", n: <<warning>>  message
```

An example warning message is shown below.

```
"rt.ada", 297: <<warning>> INLINE:  pragma has no effect
```

Error messages that begin with

```
***  Compiler Error
```

are internal compiler error messages. If any appear, they should be reported to Meridian Software Systems.

All error messages and warning messages should be self–explanatory.

19.1.7 Listings

The compiler by default does not produce listings. The **-l** option causes the compiler to produce both a listing file and an error log file. The **-lE** option causes the compiler to produce only an error log file. In the absence of these options, the compiler prints an error log to the standard output stream alone.

Listing File Contents

The listing file contains line–numbered, paginated source text with error messages and warning messages interspersed. Listing file error messages appear in the format:

```
      *
*****E error message
      *
```

Warning messages appear in the format:

```
      +
+++++W warning message
      +
```

The listing generated when **-l** is specified obeys pragma **list** and pragma **page** as described in the LRM. Pragma **list** and pragma **page** have no effect in the absence of the **-l** option.

Listing Format Control

Listing format is controlled via modifiers to the **-l** option or via a compiler default option description file named **ada.ini**. Refer to section 19.1.8 for information about the **ada.ini** file.

The **-l** option has the form **-l***modifiers*, where *modifiers* are zero or more of these letters:

c Use continuous listing format. The listing by default contains a header on each page. Specifying **-lc** suppresses both pagination and header output, producing a continuous listing.

p Obey pragma **page** directives. Specifying **-lp** is only meaningful if **-lc** has also been given. Normally **-lc** suppresses all pagination, whereas **-lcp** suppresses all pagination except where explicitly called for within the source file with a pragma **page** directive.

s Use standard output. The listing by default is written to a file with the same name as the source file and the extension **.lst**, as in **simple.lst** from **simple.ada**. Specifying **-ls** causes the listing file to be written to the standard output stream instead. This output may be redirected anywhere (e.g. to the **PRN** device).

t Generate relevant text output only. The listing by default contains the entire source program as well as interspersed error messages and warning messages. Specifying **-lt** causes the compiler to list only the source lines to which error messages or warning messages apply, followed by the messages themselves.

Any, all or none of the suffix letters **c**, **p**, **s**, and **t** may be given following **-l**, as in **-lcs**, **-lct**, or **-lcst**. The options can also be given separately, as in **-lc -ls**.

19.1.8 Default Option Description File

Compiler behavior can be modified not only by command line options, but also by a default option description file named **ada.ini**. Default options are given in **paclib\ada.ini**, while local overriding options can be given in an **ada.ini** file in the current working directory. At present, only listing format parameters can be set in a compiler default option description file.

The **ada.ini** file is an ordinary text file that may be created or edited with any editor used to edit Ada programs. The default **paclib\ada.ini** file contents are:

```
--
-- Meridian Ada compiler default option description file
--

--page_length           := 66;      -- min: 5
--top_margin            := 6;       -- min: 2
--bottom_margin         := 6;       -- min: 2
--left_margin           := 4;       -- min: 0
--page_width            := 132;     -- min: 40
--lineno_width          := 5;       -- min: 0
--marker_lines          := 1;       -- min: 0

--header_timestamp      := true;
--summary               := true;
--graphic_controls      := true;

--error_tag             := "E";
--warning_tag           := "W";
```

The **paclib\ada.ini** file as it is initially configured consists solely of comments showing examples of parameter assignments. The compiler default option description file may consist of Ada comments, blank lines, or assignments. In the example above, the initial comment characters ("--") must be deleted to make any parameter assignment effective. The comments show the default values of the parameters, so there is no need to un-comment any particular assignment unless the value is to be changed. An example of a local **ada.ini** file might be:

```
--
-- Local Meridian Ada compiler default option
-- description file
--
page_width :=  79;
-- Use all other defaults.
```

The parameters to which assignments can be made are:

bottom_margin This parameter sets the number of lines in the bottom margin of each page. The bottommost lines are blank. The minimum number of bottom margin lines that can be specified is 2. The bottom margin is suppressed by using the **-lc** (continuous listing) option.

error_tag This parameter specifies the character string that is displayed at the beginning of any error messages that occur in the listing. This string serves to highlight the error message.

graphic_controls This parameter determines how non–printable characters in the Ada source file are printed. Non–printable characters include the ASCII DEL character and all ASCII control characters except for ASCII horizontal tab and the normal line terminator characters. If this parameter's value is true, then the control characters are printed as ^*x* where *x* is the printable character derived by adding the number 64 (decimal) to the ASCII code value of the control character. For example, the ASCII BEL character, also known as Control–G (^G), is printed as ^G. Other non–printable characters are displayed as ^?. If the parameter's value is set to false, then the non–printable characters are displayed as is with no conversion.

header_timestamp This parameter specifies whether the heading information should include the date and time when the listing was generated. If its value is true, then the date and time are displayed. If its value is false, no date or time is printed.

left_margin This parameter specifies the number of blanks printed in each line before anything else (including line numbers, source lines, messages, and heading information). The left margin can be no smaller than 0 characters.

lineno_width This parameter specifies the number of characters reserved for the line number that appears to the left of each source line listed. The line number width can be no smaller than 0 characters. If 0 is specified, no line numbers are generated.

marker_lines This parameter specifies the number of marker lines to print before and after an error or warning message. Marker lines serve to make these messages stand out more from the rest of the listing. The minimum number of marker lines is 0.

page_length This parameter sets the number of lines printed per page. A page eject is placed at the end of each page. The minimum number of lines that can be specified for the page is 5. The page eject can be suppressed by using the **-lc** (continuous listing) option.

page_width This parameter specifies the number of characters in the longest line that will be printed before the line is broken to the next line. The page width can be no smaller than 40 characters. This value does not include the number of characters specified by the **left_margin** and **lineno_width** parameters.

summary This parameter specifies whether the listing should include a compilation summary at the end. If a summary is desired, this parameter's value should be set to true, otherwise it should be set to false.

top_margin This parameter sets the number of lines in the top margin of each page. Centered in the topmost lines of each page a heading is printed containing a page number, file name, and date. The minimum number of top margin lines that can be specified is 2. The top margin and heading information can be suppressed altogether by using the **-lc** (continuous listing) option.

warning_tag This parameter specifies the character string that is displayed at the beginning of any warning messages that occur in the listing. This string serves to highlight the warning message.

19.1.9 Producing Assembly Code

It is possible to produce human–readable assembly language output from a compilation by using the **-S** option to **ada**. The assembly language output file is primarily used for reference purposes. The name of the assembly language output file must be determined by using the **-l** option to **lslib**. Refer to section 19.1.4 for an explanation of the way file names are assigned. Note that the assembly language output file itself is placed in the auxiliary directory if the auxiliary directory is present.

The **-S** option must be used with care because of the way the compiler interacts with the library system. When an assembly language file is generated, but not assembled, the library is normally updated as though an object file has been produced. This means that the information in the library is incorrect with respect to the compilation unit until an object file is actually produced. If **bamp** is used prior to the creation of an actual object file, there will be unresolved symbols at link time.

One way around this problem is to use the **-fU** option. This inhibits library updates, but can be used only in certain circumstances. The **-fU** option cannot be used in compilations where specifications and bodies are co-resident in a single file. This is because compilation of a specification generates library information that is then needed for compilation of the body. This means also that when a body is present for a specification, the specification cannot be compiled with the **-fU** option. Ordinarily this is not a problem, because any code generated for a specification is replaced with the code generated for the body; the code for the body includes any code that would have been produced for the specification alone.

To summarize, the **-S** option should be used only with the body; the specification should be in a separate file. When the body is compiled with the **-S** option, the **-fU** option should be given as well.

It is also possible to correct the library information problem by assembling the assembly language output file to produce an object file. The object file produced by an assembly should be placed in the same directory in which the assembly language source file is found.

19.1.10 Examples

Example 1

Compile **x.ada** in the usual manner:

```
ada x.ada
```

Do not forget to type the extension. If you do not type the extension the **ada** program displays the error message "**missing or improper file name**".

Example 2

Compile **x.ada** with exception location maintenance code:

```
ada -fL x.ada
```

This is most useful when debugging a program that raises an exception.

Example 3

Compile **x.ada**, but with all automatic checking suppressed:

```
ada -fs x.ada
```

This is most useful after a program has been debugged and it is time to generate a "production" version of the program that is smaller and runs more quickly.

Example 4

Compile **x.ada**, but with numeric checking suppressed:

```
ada -fN x.ada
```

This retains most of the useful checks, while speeding up the program and decreasing its size.

Example 5

Compile **x.ada**, **y.ads**, and **x.adb** in the usual manner:

```
ada x.ada y.ads z.adb
```

Each source file is compiled in the order given.

Example 6

Compile `x.adb` and produce an annotated assembly language listing:

```
ada -feU -S x.adb
```

Refer to section 19.1.9 for additional information about producing assembly language output. Note that the `-fe`, `-fU`, and `-S` options typically must be used together.

Example 7

Compile `x.ada` using an alternate program library named `z.lib`:

```
ada -L z.lib x.ada
```

This presumes that `z.lib` was created in the current working directory with the `mklib` program prior to compilation.

Example 8

Compile `x.ada` verbosely, suppressing warning messages:

```
ada -fvw x.ada
```

For the `-fv` option, the compiler prints the name of each subprogram, package, or generic as it is compiled, along with the amount of symbol table space remaining. For the `-fw` option, the compiler suppresses warning messages.

Example 9

Compile `simple.ada`, producing a listing file:

```
ada -l simple.ada
```

This produces a listing file named `simple.lst` and an error log file, `simple.err`.

Example 10

Compile `simple.ada`, producing a continuous–form listing file on standard output:

```
ada -lsc simple.ada
```

This produces a listing on standard output in continuous format (no headers or pagination), as well as an error log file, `simple.err`.

Example 11

Compile `simple.ada` in a non–local directory, producing a local object file:

```
ada e:\i\me\mine\simple.ada
```

An `ada.lib` file must be present in the current directory. All output files are placed in the current directory or in the local auxiliary directory (`ada.aux`).

Example 12

Compile `simple.ada` for the 80286:

```
ada -fS simple.ada
```

Remember that a program compiled with this mode cannot be run on 8086/8088–based systems such as the original IBM PC, IBM PC–junior, IBM PC/XT, or IBM PS/2–30; the program does run on the IBM PC/AT, IBM PS/2–60, IBM PS/2–80, Zenith Z–248, and other compatible systems.

Example 13

Compile `simple.ada` running the local optimizer:

```
ada -g simple.ada
```

Runs an additional optimization pass during compilation.

Example 14

Compile `simple.ada` normally, but retain information for the global optimizer:

```
ada -K simple.ada
```

Runs the compiler normally, but does not delete `simple.int`, which can be used by a subsequent global optimization (`bamp -g`) command.

19.2 ada2

19.2.1 Invocation

```
ada2 [option ...] file.ada ...
```

19.2.2 Description

The **ada2** command is a simplified batch version of **ada** that invokes the Meridian Ada compiler. The **ada2** command allows you to compile a compilation unit that has become too large to compile with the normal **ada** program. The **ada2** command is not needed if the Meridian Ada Extended Mode Compiler is used. (The Meridian Ada Extended Mode Compiler is available separately.)

Ada2 supports all command options described for the **ada** command in section 19.1.3 except for **-g** and **-S**. Refer to section 19.1 for additional information about command line options and general behavior of the compiler.

As with **ada**, a program library must be created using **mklib** or **newlib** in advance of any compilation. The compiler aborts if it is unable to find a program library (either the default in the current working directory or as specified with the **-L** option).

The **ada2** command leaves a file named **pacout.bat** in the current working directory. The **pacout.bat** file is a temporary batch file created by **ada2** to which the **ada2** command chains to perform the actual compilation. The **pacout.bat** file may be deleted following compilation.

Note that DOS limits the number of command line arguments for batch commands to 9. The **ada2** command ignores command line arguments after the ninth.

19.2.3 Examples

Example 1

Compile **x.ada** in the usual manner:

```
ada2 x.ada
```

Example 2

Compile **x.ada**, **y.ads**, and **z.adb** in the usual manner:

```
ada2 x.ada y.ads z.adb
```

Each source file is compiled in the order given.

19.3 adaext

19.3.1 Invocation

19.3.2 Description

The **adaext** command reconfigures the compiler to run using extended memory, increasing its capacity, allowing much larger programs to be compiled. It also sets things up so that the **ramp** command can be run.

19.4 adareal

19.4.1 Invocation

19.4.2 Description

The **adareal** command reconfigures the compiler after the **adaext** command has been used to configure the compiler for using Extended Memory. This command undoes the effects of the **adaext** command.

19.5 amake

19.5.1 Invocation

```
amake [option ...] [file ...] [target ...]
```

19.5.2 Description

The **amake** program is a tool that helps to speed software development by automatically invoking compilations and linkages when source files are modified or when recompilations are required by the rules of the Ada language.

Amake treats any *file* arguments as Ada source files which it compiles unconditionally, in the order given, before building the *target(s)*. **Amake** distinguishes *file* from *target* only by the presence of a file extension in *file* (e.g. **count.ada**).

For each specified *target* compilation unit, **amake** examines the library information for source file information and dependencies. Any obsolete compilation units required by the *target* are compiled, followed by compilation of the *target* itself. A compilation unit is considered obsolete if its corresponding source file has been modified or if it has been rendered obsolete by a previous compilation. Once all compilation requirements have been satisfied, then linkage occurs, if the *target* is a main subprogram.

If no *target* is specified, then **amake** selects the most recently compiled subprogram that fits the profile for a main subprogram (i.e. a parameterless procedure).

A sequence of compilations initiated by **amake** halts if errors occur during compilation. If dependencies (the sequences of WITHs for compilation units) are changed or if new dependencies are added, then it is necessary to perform initial manual compilations to introduce the new compilation units into the library. This can be accomplished by telling **amake** the new source files or the new compilation order with appropriate *file* arguments. Subsequent recompilations then can be done with **amake**.

The **amake** program performs compilations and linkages only on units in the selected library (typically the default local library, **ada.lib**). If there are dependencies on obsolete units from non–local libraries, then **amake** complains and refuses to proceed. In that event, the non–local units must be brought up to date separately, possibly with an application of **amake**.

Compilation and linkage options can be specified in an optional *makefile*, which is a text file named **AMAKE**. The **amake** **-M** option can be used to specify an alternative *makefile* name.

A *makefile* contains lines that are comments, compilation options, or linkage options. A comment line begins with one of these character sequences:

```
#
--
```

A comment line can also be started with the keywords **rem** or **remark**.

A compilation option line begins with the keyword **compile** followed by the name of a source file and then by a sequence of options to be given to the compiler if and when that file is compiled by **amake**. If the file name is **all**, then any compilation options specified are applied to all compilations. If the file name is **others**, then any compilation options specified are applied to compilations for which no options have otherwise been specified. Compilation options are cumulative.

A linkage option line begins with the keyword **link** followed by the name of a compilation unit and then by a sequence of options to be given to the linker (**bamp**) if and when that compilation unit is linked by **amake**. If the unit name is **all**, then any linkage options specified are applied to all linkages. If the unit

name is **others**, then any linkage options specified are applied to linkages for which no options have otherwise been specified. Linkage options are cumulative.

An example of a *makefile* is:

```
-- Example makefile
--
-- All files will be compiled with -K (keep internal
-- form files around for the optimizer).
--
compile all -K
--
--
-- Suppress checks and generate a continuous listing
-- for xxx.ada.
--
compile xxx.ada -fs -lc
--
-- Generate exception location information when
-- compiling yyy.ada.
--
compile yyy.ada -fL
--
-- Perform global optimization when linking xxx.
--
link xxx -G
--
-- Generate an extended mode program for yyy named "TEST.EXP"
--
link yyy -x -o test
```

This *makefile* causes **-K** to be applied in addition to the options given for specified files. For example, these operations would take place:

```
ada -K -fs -lc xxx.ada
ada -K -fL yyy.ada
ada -K zzz.ada
bamp -G xxx
bamp -x -o test yyy
```

Note that although there is no line for **zzz.ada** in the *makefile*, the **-K** option is applied, as specified for the **all** rule.

19.5.3 Options

These options can be applied to the **amake** command line. Options supplied in the *makefile* are given to the compiler or linker, and are otherwise ignored by **amake**.

-c *compiler-program-name*
Default: as stored in program library (**ada.lib**)
The **-c** option selects an alternative compiler. The default compiler used is the one stored in the program library. The **-c** option is intended primarily for use in cross-compiler configurations, although under such circumstances, an appropriate library configuration is normally used instead.

-fflags

Default: none

The **-f** option permits one to apply **ada -f** options to the entire sequence of compilations invoked by **amake**. The list of **-f** options available are as documented for the **ada** command.

-L *library-name*

Default: **ada.lib**

Use alternate library. The **-L** option specifies an alternative name for the program library.

-M *makefile*

Default: **AMAKE**

The **-M** option selects an alternative *makefile*.

-n The **-n** option prevents the linkage step from occurring following any compilations.

19.5.4 Examples

Consider this sequence of compilation units which have already been compiled into the program library:

In file **mmm.ada**:

```
with text_io;
procedure mmm is
begin
  text_io.put_line("mmm");
end mmm;
```

In file **xxx.ada**:

```
procedure xxx is
  procedure yyy is separate;
begin
  yyy;
end xxx;
```

In file **yyy.sub**:

```
with mmm;
separate(xxx)
procedure yyy is
begin
  mmm;
end yyy;
```

Example 1

Following a modification of **xxx.ada**, apply **amake**:

```
amake xxx
```

This results in these compilations:

```
ada xxx.ada
ada yyy.sub
bamp xxx
```

The subunit **yyy** is recompiled because it becomes obsolete when its parent unit is recompiled.

Example 2

Following a modification of **mmm.ada**, apply **amake**:

```
amake xxx
```

This results in these compilations:

```
ada mmm.ada
ada yyy.sub
bamp xxx
```

The subunit **yyy** is recompiled because it becomes obsolete when the unit on which it depends, **mmm**, is recompiled.

Example 3

Wihout any modifications, apply **amake**:

```
amake mmm.ada
```

This results in these compilations:

```
ada mmm.ada
bamp mmm
```

The unit **mmm** is forcibly recompiled even though it has not been changed. The unit **mmm** is linked because **mmm.ada** contains a subprogram that fits the profile for a main subprogram (i.e. a parameterless procedure). When **amake** is not given an explicit *target*, it behaves like **bamp** with no *target* and causes the newest main subprogram to be linked. This behavior is particularly useful when compiling and linking small test programs that have not yet been entered in the library, or for forcing recompilations.

Example 4

Without any modifications, apply **amake**:

```
amake mmm.ada xxx
```

This results in these compilations:

```
ada mmm.ada
ada yyy.sub
bamp xxx
```

Now unit **mmm** is forcibly recompiled but **xxx** is the target to be remade.

19.6 auglib

19.6.1 Invocation

```
auglib [option...] library-unit [link-parameter...]
```

19.6.2 Description

The **auglib** command is used to attach additional link information to the library entry of a compilation unit. This command is primarily of use in conjunction with pragma **interface** (see Chapter 15).

The kind of information that can be attached to a library entry by **auglib** includes some linkage options and the names of non–Ada object files or code libraries. This information is used by **bamp** when it links the object file components of a program together. A subtle, but important point is that the information attached to a library entry actually applies to the low–level object linker rather than to **bamp**. Although **bamp** looks up the information, it is not used directly by **bamp**. Instead, the information is passed by **bamp** to the low–level object linker. This means that only a limited set of **bamp** options (**-o** , **-r** , **-V**) may be used with **auglib**, as these have direct analogs to object linker parameters. Most **bamp** options cannot be used to augment a library entry because those options have no meaning to the low–level object linker.

If the *library–unit* name is all, then any *link–parameter* specified with **auglib** applies to all object files linked via the **bamp** command. Note that the special name all recognized by **auglib** is a reserved word in Ada, thus there cannot be any confusion of this name with an existing library unit. The name all may appear in upper or lower or mixed alphabetic case.

Information attached to a library unit with each subsequent **auglib** command *accumulates*; new information does not replace old information. If it is necessary to delete link information, the **-r** option should first be used to remove all previous information entered with **auglib**.

The **auglib** command also provides some control over the linkage ordering of object modules. Object modules are normally linked in a default order determined by **bamp**. Linkage ordering considerations are usually important only when dealing with object code libraries as opposed to individual object code files. The default linkage order, read from left to right, is:

f_1**.obj** f_2**.obj** ... *run–time*

where f_1**.obj** files are object files corresponding to library units and *run–time* object libraries are linked last. This is essentially the syntax used by **bamp** when it invokes the low–level object linker.

Note that **bamp** invokes the low–level object linker with syntax as modified by **auglib**. The effect of an **auglib** command can be tested by running **bamp** with the option **-N** and the name of the main procedure. The **-N** option to **bamp** causes the **bamp** program to print the commands executed in the course of a "dry run" without actually performing any linkage.

19.6.3 Options

-a Link after unit. The **-a** option causes **bamp** to insert the *link–parameter* information just after the name of the object file corresponding to the *library–unit*.

-b Link before unit. The **-b** option causes **bamp** to insert the *link–parameter* information just before the name of the object file corresponding to the library–unit. Note that this is the default case.

-c "Connected" link syntax. The **-c** option causes the **bamp** command to concatenate the *link–parameter* information to the object file name, with no spaces between the two. This option is only intended for some unusual linkage situations not likely to occur on PC–DOS.

-d "Detached" link syntax. The **-d** option causes the **bamp** command to keep the *link-parameter* information separate from the object file name (no concatenation). Note that this is the default case.

-e Link at end of list. The **-e** option causes **bamp** to insert the *link-parameter* information after all other object and run-time files to be linked by the **bamp** command. This option applies only to the **all** case.

-L *library-name*
Default: **ada.lib**

Use alternate library. The **-L** option specifies the name of the program library to be used by the **auglib** program. This option overrides the default library name.

-r Remove augment information. The **-r** option causes any link information previously attached to a library unit with the **auglib** command to be disassociated with the specified library unit. This option is useful if information must be changed or deleted. If the special library name **all** is used, the **-r** option just removes the link information associated with the **all** case; information associated with particular library units is untouched.

19.6.4 Illegal Option Combinations

The **auglib** options **-a**, **-b**, and **-e** are mutually exclusive. The **-b** option is actually the default and is sufficient for most cases. The **-e** option applies only if **all** is specified as the *library-unit*.

The **-c** and **-d** options are mutually exclusive. The **-d** option is the default. The **-c** option is intended for some unusual linkage situations that are not likely to occur on PC-DOS systems.

19.6.5 Examples

Example 1

This is an example of a typical use of **auglib** in conjunction with pragma **interface**.

Consider these compilation units:

```
--Fictitious system clock interface:
package clock is
  subtype time is long_integer;
  procedure set_time(time_info: time);
    pragma interface(assembly, set_time);
end clock;
--Main procedure that calls clock interface:
with clock;
procedure clock_demo is
begin
  clock.set_time(1420);
end clock_demo;
```

In this example, an interface to an assembly language procedure called **set_time** corresponds to an Ada procedure, **clock.set_time**.

To resolve the interface in package **clock**, it is necessary to cause the assembly language object code to be linked eventually with the main program when **bamp** is invoked on **clock_demo**. This is done by augmenting the library entry of package **clock** with the name of the supplementary object file by using **auglib**. Assuming that the assembled code appears in an object file **clk.obj**, **auglib** is invoked as:

```
auglib clock clk.obj
```

This command results in an augmented library entry for **clock**, as shown in this long–format **lslib** listing (see the "**Augmented object**" line):

```
Package clock
  Source file name is "clkdemo.ada".
  Internal link name is "clo".
  Host system file name is "clock".
  Entered on Wed Apr 15  1990  17:31:01  PST.
  Last changed on Wed Apr 15  1990  17:31:01  PST.
  No spec initialization
  Augmented object:  clk.obj <obj>
  No dependencies.
  Body is optional.
```

Note that by augmenting the package in which the interface declaration occurs, any program that makes use of package **clock** is linked appropriately.

To link the program, **bamp** is simply run as usual on **clock_demo**:

```
bamp clock_demo
```

The information augmenting the library entry for package **clock** is used automatically in the linkage.

Example 2

An example of using the **–a** option with **all** is:

```
auglib -a all special.obj
```

This command causes **special.obj** to be linked after all of the Ada compilation unit object files, but before the low–level Ada run–time object files.

The corresponding linkage order is:

```
f1.obj f2.obj ... special.obj run–time
```

Example 3

An example of using the **–e** option with **all** is:

```
auglib -e all special.lib
```

This command links the file **special.lib** after the low–level Ada run–time object files.

The corresponding linkage order is:

```
f1.obj f2.obj ... run–time special.lib
```

Example 4

An example of using the **–b** option is:

```
auglib -b clock clk.obj
```

This command links the file **clk.obj** before the object module corresponding to the library unit clock.

The corresponding linkage order is:

```
f1.obj ... clk.obj clock.obj f1.obj ... run–time
```

Recall that **–b** is the default case.

Example 5

An example of using the **-a** option is:

```
auglib -a clock clk.obj
```

This command links the file **clk.obj** after the object module corresponding to the library unit clock.

The corresponding linkage order is:

```
f1.obj ... clock.obj clk.obj f1.obj ... run-time
```

Example 6

An example of applying the **bamp -V** option permanently to a program is:

```
auglib ginormous -V d:\tmpfile
```

This causes the program **ginormous** to be linked always in "virtual mode" when **bamp** is run on it. The argument to **-V** is just an example of a scratch file name to supply. This option is useful for linking large programs that cause the low–level object linker to run out of memory.

An important note for applying the **-V** option to the low–level object linker is that it must appear first, before any object files. This is the default case, but if other augment information is given, it is important to remember not to permit it to appear before the **-V** option.

19.7 bamp

19.7.1 Invocation

```
bamp [option ...] [main-procedure-name]
```

19.7.2 Description

The **bamp** (Build Ada Main Program) command creates an executable program given the name of the main subprogram. The *main-procedure-name* given to **bamp** must be a parameterless procedure that has already been compiled.

Note: Be careful not to confuse the name of the source file containing the main subprogram (e.g. **simple.ada**) with the actual name of the main subprogram (e.g. **simple**).

If a *main-procedure-name* is not specified on the **bamp** command line, **bamp** links using the last-compiled subprogram that fits the profile for a main subprogram. To determine which subprogram will be used when no main subprogram is given to **bamp**, use the **lslib -t** option. When in doubt, it may be best to specify the main subprogram explicitly.

Note that when no main subprogram is specified, **bamp** selects the most recently compiled subprogram, not the most recently linked subprogram. If several different main subprograms are linked between compiles, still the most recently compiled subprogram is selected if no subprogram is explicitly specified.

The **bamp** program functions as a high-level linker. It works by creating a top-level main program that contains all necessary context clauses and calls to package elaboration procedures. The main program is created as an internal form file on which the code generator is run. Following this code generation pass, all the required object files are linked.

Unless the **-x** option is given, the **bamp** command produces a standard DOS Real Mode program. The **-x** option produces an Extended Mode program (see Chapter 11 for a discussion of Extended Mode Meridian Ada).

An optional optimization pass can be invoked via the **bamp** command. The details of optimization are discussed in Chapter 7. The **bamp** options relevant to optimization, **-g** and **-G** , are discussed below.

Programs compiled in Debug mode (with the **ada -fD** option) are automatically linked with the Meridian Ada source level debugger.

19.7.3 Options

-A Aggressively inline. This option instructs the optimizer to aggressively inline subprograms when used in addition to the **-G** option. Typically, this means that subprograms that are only called once are inlined. If only the **-G** option is used, only subprograms for which pragma **inline** has been specified are inlined.

-c *compiler-program-name*

Default: As stored in program library.

Use alternate compiler. The **-c** option specifies the complete (non relative) directory path to the Meridian Ada compiler. This option overrides the compiler program name stored in the program library. The **-c** option is intended for use in cross-compiler configurations, although under such circumstances, an appropriate library configuration is normally used instead.

-f Suppress main program generation step. The **-f** option suppresses the creation and additional code generation steps for the temporary main program file. The **-f** option can be used when a simple

change has been made to the body of a compilation unit. If unit elaboration order is changed, or if the specification of a unit is changed, or if new units are added, then this option should not be used. The **-f** option saves a few seconds, but places an additional bookkeeping burden on you. The option should be avoided under most circumstances. Note that invoking **bamp** with the **-n** option followed by another invocation of **bamp** with the **-f** option has the same effect as an invocation of **bamp** with neither option (**-n** and **-f** neutralize each other).

-g Perform global optimization only. The **-g** option causes **bamp** to invoke the global optimizer on your program. Compilation units to be optimized globally must have been compiled with the **ada -K** option.

-G Perform global and local optimization. The **-G** option causes **bamp** to perform both global and local optimization on your program. This includes performing pragma **inline**. As with the **-g** option, compilation units to be optimized must have been compiled with the **ada -K** option.

-i The **-i** option is used in conjunction with the **bamp -r** option when producing "pre-linked" code for use with the Intel Development Tools. The **-i** option causes certain information to be emitted into the object file that is needed under some circumstances by the Intel linker, **LINK86**. By default, pre-linked object modules use the Microsoft object format.

-I Link the program with a version of the tasking run-time which supports pre-emptive task scheduling. This option produces code which handles interrupts more quickly, but has a slight negative impact on performance in general.

-L *library-name*

Default: **ada.lib**

Use alternate library. The **-L** option specifies the name of the program library to be consulted by the **bamp** program. This option overrides the default library name.

-m Produce link map. The **-m** option causes a text file containing a link map to be written. The link map is Microsoft-compatible and the link map file name has the extension **.map** for Real Mode programs (the default). For Extended Mode Programs (produced when the **bamp -x** option is given), the link map is OS/x86-compatible and the link map file name has the extension **.xmp**.

-M *main-program-stack-size*

Default:

- 20K in Real Mode programs
- 64K in Extended Mode programs, when tasking is not used
- 64K- <task-stack-size> in Extended Mode programs, when tasking is used

Set main program stack size. The **-M** option sets the stack size (number of decimal bytes) for the main program (excluding tasking). Note that the sum of the main program stack size and the tasking stack size must be no more than 64K bytes.

-n No link. The **-n** option suppresses actual object file linkage, but creates and performs code generation on the main program file. Note that invoking **bamp** with the **-n** option followed by another invocation of **bamp** with the **-f** option has the same effect as an invocation of **bamp** with neither option. That is, **-n** and **-f** neutralize each other.

-N No operations. The **-N** option causes the **bamp** command to do a "dry run"; it prints out the actions it takes to generate the executable program, but does not actually perform those actions. The same kind of information is printed by the **-P** option.

-o *output-file-name*

Default: *file.exe*

Use alternate executable file output name. The **-o** option specifies the name of the executable program file written by the **bamp** command. This option overrides the default output file name.

-P Print operations. The **-P** option causes the **bamp** command to print out the actions it takes to generate the executable program as the actions are performed.

-r Create re-linkable output. The **-r** option causes an object file (**.obj** file) to be generated rather than an executable file (**.exe** file). The resulting file contains all symbol and relocation information, and can then be used with any low-level linker accepting object files compatible with the Intel or Microsoft object formats.

-s *task-stack-size*

Default:

- 20K if tasking used
- Zero if tasking not used

Use alternate tasking stack size. The **-s** option specifies the number of bytes (in decimal) to be allocated to all the tasks to be activated in the Ada program. This option overrides the default task stack size. Note that the sum of the main program stack size and the tasking stack size must be somewhat smaller than 64K bytes. The size of individual task activation stacks can be specified with a length clause, described in Chapter 6.

-u Link software floating point library. Use of the **-u** option enables a program containing floating point computations to run with or without a math co-processor chip. A related compiler option, the **ada -fF** option, also can be used to control the action of the run-time in the absence of a math co-processor chip. The ada **-fF** option and the **bamp -u** option should not both be used in the same program.

-v Link verbosely. The **-v** option causes the **bamp** command to print out information about what actions it takes in building the main program such as:

- The name of the program library consulted.
- The library search order (listed as "saves" of the library units used by the program).
- The name of the main program file created (as opposed to the main procedure name).
- The elaboration order.
- The total program stack size.
- The name of the executable load module created.
- The verbose code generation for the main program file.

-V *scratch-file*

Link using "virtual" mode. This option allows larger programs to be linked, although slightly more slowly. A *scratch-file* must be specified. The *scratch-file* can reside on a RAM disk (if one is available) for faster operation. The **-V** option affects only the operation of the low-level object linker. The *scratch-file* is used as scratch memory in which the various object files are linked.

Note: The **-V** option can be supplied once to **auglib** to make it a permanent effect for a program. An example of this is given in section 19.6.

-W Suppress warnings. This option allows you to suppress warnings from the optimizer.

-x The **-x** option is used to create an Extended Mode program. This option applies only to Extended Mode Meridian Ada. The **-x** option produces a program that can be run with the **xamp** command to run in Extended Mode (a **.exp** file). If the **-x** option is not used, a Real Mode program (a **.exe** file) is produced.

19.7.4 Examples

Example 1

Link main subprogram **simple** in the usual manner:

```
bamp simple
```

Example 2

Link the most recently compiled main subprogram:

```
bamp
```

When no subprogram is specified, the **bamp** program displays the name of the library unit being linked as the main subprogram. Note that **bamp** selects the most recently compiled subprogram, not the most recently linked subprogram.

Example 3

Link main subprogram **simple**, creating an executable load module named **simp.exe** instead of **simple.exe**:

```
bamp simple -o simp
```

Example 4

```
bamp -f simple
```

Be very cautious in using the **-f** option. The object file created by **bamp** corresponding to the main program is always named **at.obj** ("Ada top"). If **bamp** is invoked on another main subprogram (say, **sieve**, for example) then **at.obj** reflects the contents of the **at.int** file created and compiled for that program. If the **-f** option is subsequently used to link **simple**, then the wrong **at.obj** file is used in the object linkage.

Example 5

Link the main subprogram **simple** occurring in an alternate library, **z.lib**:

```
bamp -L z.lib simple
```

Example 6

Link the main subprogram **reschedule_demo** using an alternate value for the main program stack size:

```
bamp -M 10240  reschedule_demo
```

This example specifies that 10K bytes are to be reserved for the main program stack instead of the default 20K. Note that this value does not affect the amount of space reserved for the task stack space, which is determined by the **-s** option.

Example 7

Link the main subprogram **reschedule_demo** using an alternate value for the total task stack space:

```
bamp -s 30720  reschedule_demo
```

This example specifies that 30K bytes are to be reserved for the total task stack space instead of the default 20K. Note that this value does not affect the amount of space reserved for the main program stack.

Do not use the `-s` option to set the amount of task stack space to zero when no tasking is used in a program; the `bamp` program determines this situation automatically and then reserves no additional stack space for tasking.

Example 8

Link the main subprogram `reschedule_demo` using alternate values for both the total task stack space and the main program stack space:

```
bamp -s 30720 -M 10240 reschedule_demo
```

This example reserves 30K for the task stack space and 10K for the main program stack space. Note that the total stack space (the sum of the task stack space and the main program stack space) reserved in this way presently cannot exceed 64K.

Example 9

Create main program object file, `at.obj`, for the main subprogram `simple` but do not link the object files in the program:

```
bamp -n simple
```

Example 10

Do a "dry run" of the linkage operations for the main subprogram `simple`, printing out the system commands that would be used to perform the linkage:

```
bamp -N simple
```

This is useful to determine the linkage order of the various object files and object libraries in the program. The `-N` option can be used to "debug" `auglib` modifications.

Example 11

Link the main subprogram `simple` without creating an executable load module:

```
bamp -r simple
```

This option is useful for creating an object module to be linked with a different object linker (e.g. the DOS system linker). The final object file created in this example is `simple.obj`.

Example 12

Link the main subprogram `simple` verbosely:

```
bamp -v simple
```

This prints out information about the elaboration process and the files created.

Example 13

Link the main subprogram `simple`, but print out the system commands used to do so as the commands are performed:

```
bamp -P simple
```

The information printed is the same as that printed for the `-N` option.

Example 14

Link the main subprogram **simple** using a different **ada** command with which to generate the main program object file (**at.obj**):

```
bamp -c c:\zada\bin\zada.exe simple
```

The **-c** option is intended for use in cross–compiler configurations, although under such circumstances, an appropriate library configuration is normally used instead.

Example 15

```
bamp -V d:\tmpfile ginormous
```

This is useful when a program becomes too large to link under normal circumstances. Note that if drive **d:** is a RAM disk in this example, virtual mode linkage can go faster; however, the RAM disk must be large enough to accommodate all the object modules used to generate the executable load module.

Example 16

Link the main subprogram **ginormous** to produce an Extended Mode program:

```
bamp -x ginormous
```

This can be done only with Extended Mode Meridian Ada. The resulting unbound Extended Mode program, **ginormou.exp**, must be run with the **ramp** command.

Example 17

Link the main subprogram **simple** to produce a standard DOS Real Mode program, and produce a symbol map:

```
bamp -m simple
```

This produces a Real Mode map file, **simple.map**, and a Real Mode program, **simple.exe**.

Example 18

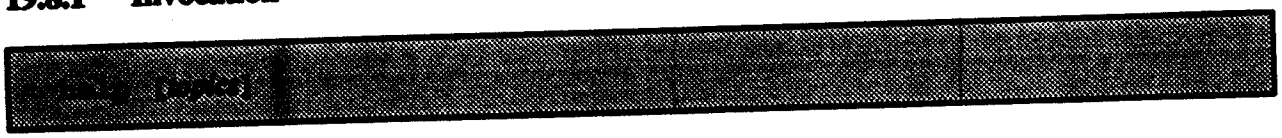
Link the main subprogram **ginormous** to produce an Extended Mode program, and produce a symbol map:

```
bamp -m -x ginormous
```

This can be done only if Extended Mode Meridian Ada has been installed. An Extended Mode map file, **ginormou.xmp**, and an unbound Extended Mode program, **ginormou.exp**, are produced. The Extended Mode program must be run with the **ramp** command. The map file can be examined with the DOS **type** command.

19.8 help

19.8.1 Invocation



19.8.2 Description

The `help` command provides helpful hints and some troubleshooting information about the selected topic. If no topic is given on the command line, the `help` command displays the list of available topics.

19.9 lnlib

19.9.1 Invocation

```
lnlib [option ...] link-library-name...
```

19.9.2 Description

The **lnlib** command adds library references, called links, to a program library. Once a program library has been created using **mklib**, library references (links) must normally be added using **lnlib** so that the standard packages (e.g. **text_io**) may be introduced into compilations that use the new library. The **newlib** command normally adds default links so that explicit use of **lnlib** is unnecessary except when building complicated programming project directories.

Each link added to a library is used in extended library searches by the compiler and by **bamp** (see Chapter 8).

The *link–library–name* given to **lnlib** should be a full file path specification. Relative directory specifications, e.g. "..\" should not normally be used because searches through such links are performed relative to the current working directory. If the specified library is itself the target of further library linkages, any relative directory references change as the current working directory is changed.

The **lnlib** command can also un–link libraries from a program library if the **-r** option is used.

The **-k** option to the **lslib** command can be used to list link entries.

19.9.3 Options

-L *library–name*

Use alternate library. Default: **ada.lib**

The **-L** option specifies an alternative name for the program library.

-r *library–name* [*replacement–library–name*]

Replace link. The option **-r** replaces or deletes library links from a program library. Note that this is generally not a good idea because some library units can be left with dangling library references. It is best to determine what library units depend on what other libraries (by using the **lslib** command) before removing library links.

The **-r** option changes the syntax of **lnlib** slightly so that only one or two *library–name* parameters are accepted on the command line (normally, **lnlib** takes an indefinite number of library name parameters).

Note: Only link entries are affected by the **-r** option; no program libraries are deleted.

19.9.4 Examples

Example 1

Add a link entry into the program library:

```
lnlib c:\ada\paclib\ada.lib
```

A subsequent invocation of **lslib** with the **-k** option prints any link entries; this might show:

```
Links to:
 c:\ada\paclib\ada.lib
```

Example 2

Add several link entries into the program library:

```
lnlib c:\ada\dosenv\ada.lib c:\ada\adautil\ada.lib
```

A listing of the resulting links might then show:

```
Links to:
  c:\ada\adautil\ada.lib
  c:\ada\dosenv\ada.lib
  c:\ada\paclib\ada.lib
```

Example 3

Remove a library link entry:

```
lnlib -r e:\ada\utils\ada.lib
```

Example 4

Replace a library link entry for `e:\ada\utils\ada.lib` with `e:\ada\adautil\ada.lib`:

```
lnlib -r e:\ada\utils\ada.lib e:\ada\adautil\ada.lib
```

This deletes the link entry for `e:\ada\utils\ada.lib`.

A subsequent run of `lslib` with the `-k` option might show:

```
Links to:
 e:\ada\adautil\ada.lib
 e:\ada\paclib\ada.lib
```

Example 5

Add a link entry to an alternate library, `z.lib`:

```
lnlib -L z.lib c:\ada\paclib\ada.lib
```

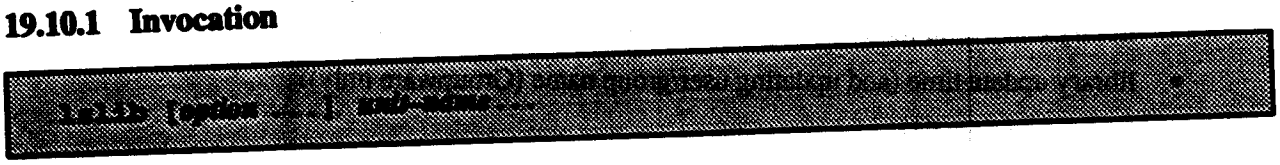
Example 6

Delete a link entry from an alternate library, `z.lib`:

```
lnlib -L z.lib -r e:\ada\utils\ada.lib
```

19.10 lslib

19.10.1 Invocation



19.10.2 Description

The **lslib** command prints information about the contents of a program library or about a particular entry for a compilation unit in the library.

When invoked without options, **lslib** simply lists the units in the default program library.

In order for any Groupware information to be listed, the program library (default or via **-L** option) must be a Groupware library.

19.10.3 Options

-a List all, short. List the "immediate" library sub–tree used in context lookups by the compiler (to two levels). This option is normally given alone; if a *unit–name* is also specified, then only information about that compilation unit is displayed.

-A List all, long. List entire library graph (to indefinite levels). This option is normally given alone; if a *unit–name* is also specified, then only information about that compilation unit is displayed.

-d *unit*
List dependencies. List all units on which *unit* depends (i.e. everything that appears in a context clause (with clause) of the specified unit). This applies only to dependencies on compilation units in the default or specified library.

-h List "header" information. Information listed includes:

- library internal name string (Groupware only)
- library documentation string (Groupware only)
- library version stamp
- target machine type
- library creation time (and creating user/group name (Groupware only))
- last library update time (and updating user/group name (Groupware only))
- library lock time and locking user/group name, if locked (Groupware only)
- last unique link symbol generated
- last unique file name generated
- default compiler name
- auxiliary directory name (if any)
- archive name (if any)
- optons

-k List link entries: The **-k** option lists all link entries in the default or specified library.

-l Long-format listing. Information listed includes:

- source file name
- library entry time (and entering user/group name (Groupware only))
- library update time (and updating user/group name (Groupware only))
- list of dependencies

This option is equivalent to specifying **-o** with all modifiers.

-L *library-name*

Use alternate library. Default: **ada.lib**

The **-L** option specifies an alternative name for the program library.

-o *modifiers*

Additional information listing, as specified by one or more of the following modifiers (the modifiers must be separated from **-o** by at least one blank):

b Indicates if the body is defined and information about subunits, if any.

d Lists dependencies.

f Lists source file name.

i Lists mapping from programmer-assigned name to unique link symbol and unique base file name.

m Lists unit type (subprogram, main subprogram, package, or task body), an indication if the unit is obsolete (needs recompilation), an indication if the unit uses tasking or initialization code, an indication if the unit uses debugging code, and any link options which may have been added to an entry using **auglib**.

t Lists library entry time (and entering user/group name (Groupware only)) and library update time (and updating user/group name (Groupware only)).

-t List in reverse time order. List units in order from most recent to least recent time of modification (or entry into the library).

-w *unit* List dependents. List everything that depends on the specified *unit* (i.e. everything that names *unit* in a context clause).

Options to the **lslib** program can be combined, as in **-hkl**. This is equivalent to specifying the options separately, e.g. "**-h -k -l**".

19.10.4 Examples

Example 1

List the compilation unit entries in the program library:

```
lslib
```

This invocation of **lslib** might produce a listing like:

```
Package clock
Subprogram clock_demo
Package direct_int_io
Subprogram file_form_demo
Package io_example
Subprogram reschedule_demo
```

Example 2

List the information in long format for a particular compilation unit entry:

```
lslib -l reschedule_demo
```

This invocation of `lslib` might produce a listing like:

```
Subprogram reschedule_demo
  Source file name is "td.ada".
  Entered on Tue Apr 14 1987 12:39:55 PST.
  Changed on Tue Apr 14 1987 16:26:18 PST.
  Dependent on: text_io
```

Example 3

List the link entries (as well as the compilation unit entries):

```
lslib -k
```

This invocation of `lslib` might produce a listing like:

```
Links to:
  c:\ada\adautil\ada.lib
  c:\ada\paclib\ada.lib

Package clock
Subprogram clock_demo
Package direct_int_io
Subprogram file_form_demo
Package io_example
Subprogram reschedule_demo
```

Example 4

List all units on which `text_io` depends:

```
lslib -d text_io
```

The result of this invocation of `lslib` is:

```
Package io_exceptions
Package tcd_runtime
```

Example 5

List all units depending on example package clock:

```
lslib -w clock
```

The result of this invocation of `lslib` is:

```
Subprogram clock_demo
```

Example 6

List all units in an alternate library:

```
lslib -L c:\ada\paclib\ada.lib
```

This is an example of listing the contents of the distribution library. Note that your distribution library might be elsewhere.

This is likely to display something very similar to:

```
Package ada_io
Package calendar
Package dcd_runtime
Generic Package direct_io
Package enum_io_runtime
Package file_manage
Package fio
Package fixed_io_runtime
Package float_io_runtime
Package iio
Package int_io_runtime
Package io_exceptions
Package machine_code
Package num_io_runtime
Generic Package sequential_io
Package syerr
Package syio
Package system
Package sytime
Package task_control
Package tcd_runtime
Package terminate_runtime
Package text_io
Package tioc_runtime
```

Example 6

List the library header:

```
lslib -h
```

In this example you can see the information displayed by a Groupware version of **lslib**. Note that the library has been locked in this particular case and that there is a documentation string (**lslib examples**) present.

```
ada.lib:
lslib examples

Library version 10.
Generated for target i8086 msdos.
Library created on Thu Aug 31 1989 17:14:32 by <gordon, r&d>.
Library updated on Fri Sep 15 1989 15:10:48 by <bill, r&d>.
Library locked on Fri Sep 15 1989 18:11:08 by <gordon, r&d>.
Last linker name used "aaaaaaaa".
Last file name used "aaaaaaaa".
Default compiler name "c:\ada\bin\ada.exe".
Config file "c:\ada\paclib\intel.cfg".
Aux file directory name "ada.aux\".
```

Example 7

List additional information:

```
lslib -o bim reschedule_demo
```

This invocation of `lslib` produces a listing of information that is not available in the `-L` option, as follows:

```
Subprogram reschedule_demo
  Internal linkname is "reschedu".
  Host system file name is "reschedu".
  Can be main program.
  Uses tasking
  Dependent on: text_io
  Body is defined.
```

19.11 mklib

19.11.1 Invocation

```
mklib [option ...]
```

19.11.2 Description

The **mklib** command creates a new program library. When options are not specified, all the defaults described below apply. Some caution is required when using **mklib** because it is possible to create libraries that supply completely nonsensical information to the compiler or the linker (e.g. a non–existent auxiliary directory, an incorrect compilation command, etc.).

Note that the **newlib** command is normally the first resort; it is usually unnecessary to use **mklib** except when building complicated programming project directories.

Once a program library has been created using **mklib**, library references (links) must normally be added using **lnlib** so that the standard packages (e.g. **text_io**) can be introduced into compilations that use the newly–created library. The **newlib** command adds some default links.

To summarize, the steps involved in building a program library with **mklib** are:

1. Create the auxiliary directory (if one is to be specified with the **mklib** command) using the **mkdir** command.
2. Create the library using **mklib**.
3. Enter into the library any desired library links using **lnlib**.

19.11.3 Options

–a *auxiliary–directory*\

Default: None (current working directory).

Name auxiliary directory. The specified auxiliary directory is to be associated with the created program library. All object files, interface description files, and other files created by compilations related to the library are to be kept in the auxiliary directory. The **mklib** command does not automatically create the auxiliary directory; it must either exist or be created with the PC–DOS **mkdir** command prior to the first compilation involving the library.

Note: The auxiliary directory name must be terminated with a slash ("\").

–c *compiler–program–name*

Default: **ada**

Name compiler. The **–c** option specifies the name of the Meridian Ada compiler. This information is saved in the program library so that the **bamp** command can find the appropriate compiler to use.

–L *library–name*

Default: **ada.lib**

Use alternate library. The **–L** option specifies an alternative name for the program library. If an alternative name is given, all operations involving the library (including compilation) must specify that name. Normally, only one program library can exist in a single directory, unless a separate auxiliary directory is associated with each co–resident library.

19.11.4 Examples

Example 1

Create a library with the default name and for the default target, but with no auxiliary directory:

```
mklib -c c:\ada\bin\ada.exe
```

Note that this example assumes the installation directory was `c:\ada`.

Example 2

Create a library with the default name, for the default target, with an auxiliary directory:

```
mklib -c c:\ada\bin\ada.exe -a ada.aux\
```

Note that this example assumes that the installation directory was `c:\ada`. Please note the terminating slash ("\") on the name of the auxiliary directory. The slash is required; otherwise, the compiler and `bamp` build incorrect path names for associated files.

Recall that `mklib` does not create the auxiliary directory; that must be done separately with:

```
mkdir ada.aux
```

Example 3

Create a library with an alternate name, for the default target, with an auxiliary directory:

```
mklib -L z.lib -c c:\ada\bin\ada.exe -a z.aux\
```

Note that this example assumes that the installation directory was `c:\ada`. Also note the terminating slash ("\") on the name of the auxiliary directory. The slash is required; otherwise, the compiler and `bamp` build incorrect path names for associated files.

Recall that `mklib` does not create the auxiliary directory; that must be done separately with:

```
mkdir z.aux
```

All Meridian Ada library commands (including `ada`) that need to reference this library must do so explicitly; all commands use the same `-L` option for this. For example, to compile `simple.ada` using `z.lib` instead of `ada.lib`, this command would be given:

```
ada -L z.lib simple.ada
```

An example of a subsequent `lslib` command using `z.lib` is:

```
lslib -L z.lib
```

This `lslib` command would display:

```
Subprogram simple
```

19.12 modlib

19.12.1 Invocation

```
modlib [option ...]
```

19.12.2 Description

The **modlib** command can be used to help manage a programming project's libraries. With it you can arbitrarily lock a library against update or insert additional documentation information in the library header.

Note: This is a Groupware feature.

19.12.3 Options

-d *string*
: Change the internal documentation string. Up to 255 characters can be placed in the library header and is displayed immediately below the library name when the header is listed. To remove it simply change it to a null (empty) string.

-l
: Lock the library. When the library is locked it cannot be used as a program library by the compiler, **ada**. However, **bamp** can still build programs from a locked library and the library utilities can still be used to modify a locked library. The header contains the name and group of the user (if available) that locked the library and can be displayed using **lslib -h**. Note that this lock is independent of the library sharing mechanism built into Groupware.

-L *library-name*
: Default: **ada.lib**

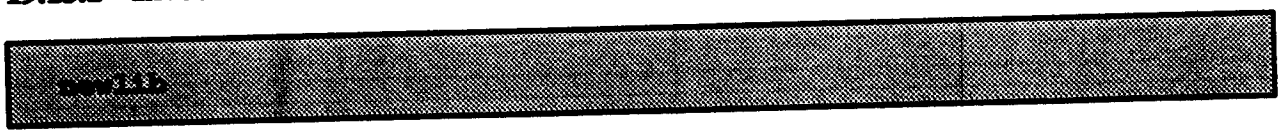
Use alternate library. The **-L** option specifies the name of the program library upon which the **modlib** command is to operate. This option overrides the default library name.

-n *string*
: Change the internal library name. A name of up to 255 characters can be placed in the library header. This internal name is for information only and cannot be specified as the library name for a **-L** option. It is displayed in place of the external name when the header is listed. To default to the external name simply change the internal name back to a null (empty) string.

-u
: Unlock the library. This restores the library to its original (unlocked) status. The ability to unlock the library is not restricted to the user that locked it.

19.13 newlib

19.13.1 Invocation



19.13.2 Description

The **newlib** command creates a program library in the current working directory with the following useful default characteristics:

- the default name, **ada.lib**
- an auxiliary directory named **ada.aux**
- the default compiler name

In addition to these default characteristics, a library link is established to the standard library area so that the standard units (e.g. **text_io**) can be used in compilations. Also, the auxiliary directory, **ada.aux**, is created. Finally, **newlib** lists the contents of the newly created program library.

The **newlib** program is actually a command file that can be modified to create libraries with any appropriate defaults. Normally, **newlib** is modified at installation time to conform to any site–dependent defaults. For example, **lnlib** commands are added to **newlib.bat** for any standard packages installed on the system, such as the DOS Environment Library or the Meridian Ada Utility Library.

The **mklib** and **lnlib** programs should be used when building libraries with any unusual requirements.

Output from **newlib** looks something like this:

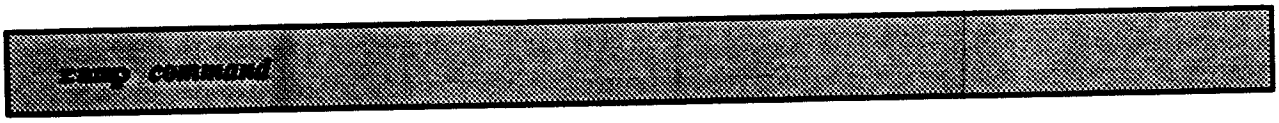
```
ada.lib:
Library version 9.
Generated for target i8086.
Last updated on Wed Apr 15  1989  17:31:02  PST.
Last linker name used "aaa".
Last file name used "aaaaaaaa".
Default compiler name "c:\ada\bin\ada.exe".
Aux file directory name "ada.aux\".

Links to:

 c:\ada\paclib\ada.aux
```

19.14 ramp

19.14.1 Invocation



19.14.2 Description

The **ramp** program is used to run unbound Extended Mode Meridian Ada programs. The command form for running an Extended Mode program with **ramp** is identical to that for running a Real Mode program, except that **ramp** precedes the rest of the command line:

ramp ***program-name*** [*options*...] [*I/O-redirections*]

The *program-name* is the name of an Extended Mode program that can be found anywhere in the DOS search path (the search path is shown with the DOS **path** command). *Options* are whatever command line options apply to the Extended Mode program to be run. I/O-redirections are standard DOS command line I/O redirections. Examples are given below.

If necessary, the **ramp** command may be hidden by use of DOS batch (**.bat**) files or eliminated entirely by processing an Extended Mode program with the separately available bind program. A brief discussion of the bind program is given in section 11.4.3.

Before the **ramp** program is used to load and run an Extended Mode program the **adaext** program must be run to properly set up the Extended Mode environment.

19.14.3 Files Used by Ramp

Extended Mode Meridian Ada program files that are run by **ramp** have the extension **.exp**. Real Mode Meridian Ada program files have the extension **.exe**. Both kinds of programs are produced as the final products of the **bamp** command. The **bamp -x** option is used to produce Extended Mode programs with the **.exp** extension.

19.14.4 Examples

Example 1

Run an Extended Mode program:

```
ramp extp
```

This example searches for a program file named **extp.exp** anywhere in the DOS command search path and runs the program in Extended Mode.

Example 2

Run an Extended Mode program with command line arguments:

```
ramp extp -o file.out
```

This example runs the program file named **extp.exp** with command line arguments that apply to the **extp** program.

Example 3

Run an Extended Mode program with output redirection:

```
ramp extp > PRN
```

This example runs the program file named **extp.exp** and redirects the output to the printer.

19.15 rmlib

19.15.1 Invocation

```
rmlib [option ...] [unit-name ...]
```

19.15.2 Description

The **rmlib** command deletes a library unit entry from the specified program library. When an entry has been removed from a program library, all files associated with that library unit are deleted (object file, interface description file, etc.), except for the source file. Source files (**.ada** files) are never deleted by **rmlib**.

The **rmlib** command does not permit units to be deleted arbitrarily. A unit entry cannot be deleted if other units depend on it. By deleting or appropriately updating the entries for other dependent units, it becomes possible to delete an entry. For example, if unit **a** withs **b**, the entry for unit **b** cannot be removed unless the entry for **a** is first deleted or updated after deleting the context clause for **b**.

19.15.3 Options

-a Remove all. The **-a** option deletes all compilation unit entries in the library, but leaves the library intact. Link entries are preserved. When the **-a** option is used, no *unit–name* parameter should be given.

-b Remove body information. The **-b** option deletes only information about the body of the unit, not information about the specification.

-L *library–name*

Default: **ada.lib**

Use alternate library. The **-L** option specifies the name of the program library upon which the **rmlib** program is to operate. This option overrides the default library name.

-r Remove "recursively". The **-r** option deletes all local entries on which the unit depends as well as the unit entry itself. Only entries in the local library are affected; libraries with link entries in the local library are unaffected (e.g. **text_io** does not disappear when **-r** is used).

-v Remove verbosely. The **-v** option lists each file related to the library entry as the file is deleted.

19.15.4 Examples

Example 1

Delete the entry for unit **simple**:

```
rmlib simple
```

Example 2

Delete all entries verbosely, leaving the library otherwise intact:

```
rmlib -a -v
```

Example 3

Delete the entry for unit **simple** from a different library:

```
rmlib -L c:\ada\test\ada.lib simple
```

Example 4

Delete information about the body of unit **simple**, leaving information about the specification of **simple** intact:

```
rmlib -b simple
```

Appendix F Implementation–Dependent Characteristics

This appendix lists implementation–dependent characteristics of Meridian Ada. Note that there are no preceding appendices. This appendix is called *Appendix F* in order to comply with the Reference Manual for the Ada Programming Language* (LRM) ANSI/MIL–STD–1815A which states that this appendix be named Appendix F.

Implemented Chapter 13 features include length clauses, enumeration representation clauses, record representation clauses, address clauses, interrupts, package **system**, machine code insertions, pragma **interface**, and unchecked programming.

F.1 Pragmas

The implemented pre–defined pragmas are:

elaborate	See the LRM section 10.5.
interface	See section F.1.1.
list	See the LRM Appendix B.
pack	See section F.1.2.
page	See the LRM Appendix B.
priority	See the LRM Appendix B.
suppress	See section F.1.3.
inline	See the LRM section 6.3.2. This pragma is not actually effective unless you compile/link your program using the global optimizer.

The remaining pre–defined pragmas are accepted, but presently ignored:

controlled	**optimize**	**system_name**
shared	**storage_unit**	
memory_size		

Named parameter notation for pragmas is not supported.

When illegal parameter forms are encountered at compile time, the compiler issues a warning message rather than an error, as required by the Ada language definition. Refer to the LRM Appendix B for additional information about the pre–defined pragmas.

F.1.1 Pragma Interface

The form of pragma **interface** inMeridian Ada is:

pragma interface (*language*, *subprogram* [, *"link–name"*]);

where:

language This is the interface language, one of the names **assembly**, **builtin**, **c**, **microsoft_c**, or **internal**. The names **builtin** and **internal** are reserved for use by Meridian compiler maintainers in run–time support packages.

subprogram This is the name of a subprogram to which the pragma **interface** applies.

*All future references to the Reference Manual for the Ada Programming Language appear as the LRM.

link–name This is an optional string literal specifying the name of the non–Ada subprogram corresponding to the Ada subprogram named in the second parameter. If *link–name* is omitted, then *link–name* defaults to the value of *subprogram*. Depending on the language specified, some automatic modifications may be made to the *link–name* to produce the actual object code symbol name that is generated whenever references are made to the corresponding Ada subprogram. The object code symbol generated for *link–name* is always translated to upper case. Although the Meridian object linker is case–sensitive, it is a rare object module that contains mixed–case symbols; at present, all Meridian 80x86 object modules use upper case only.

It is appropriate to use the optional *link–name* parameter to pragma **interface** only when the interface subprogram has a name that does not correspond at all to its Ada identifier or when the interface subprogram name cannot be given using rules for constructing Ada identifiers (e.g. if the name contains a '$' character).

The characteristics of object code symbols generated for each interface language are:

assembly The object code symbol is the same as *link–name*.

builtin The object code symbol is the same as *link–name*, but prefixed with two underscore characters ("__"). This language interface is reserved for special interfaces defined by Meridian Software Systems, Inc. The **builtin** interface is presently used to declare certain low–level run–time operations whose names must not conflict with programmer–defined or language system defined names.

c The object code symbol is the same as *link–name*, but with one underscore character ('_') prepended. This is the convention used by the C compiler.

internal No object code symbol is generated for an **internal** language interface; this language interface is reserved for special interfaces defined by Meridian Software Systems, Inc. The **internal** interface is presently used to declare certain machine–level bit operations.

microsoft_c The object code symbol is the same as *link–name*, but with one underscore character ('_') prepended. This is the convention used by the Microsoft C compiler.

The low–level calling conventions are changed only in the case of a **microsoft_c** interface. No automatic data conversions are performed on parameters of any interface subprograms. It is up to the programmer to ensure that calling conventions match and that any necessary data conversions take place when calling interface subprograms.

A pragma **interface** may appear within the same declarative part as the subprogram to which the pragma **interface** applies, following the subprogram declaration, and prior to the first use of the subprogram. A pragma **interface** that applies to a subprogram declared in a package specification must occur within the same package specification as the subprogram declaration; the pragma **interface** may not appear in the package body in this case. A pragma **interface** declaration for either a private or nonprivate subprogram declaration may appear in the private part of a package specification.

Pragma **interface** for library units is not supported.

Refer to the LRM section 13.9 for additional information about pragma **interface**.

F.1.2 Pragma Pack

Pragma **pack** is implemented for composite types (records and arrays).

Pragma **pack** is permitted following the composite type declaration to which it applies, provided that the pragma occurs within the same declarative part as the composite type declaration, before any objects or components of the composite type are declared.

Note that the declarative part restriction means that the type declaration and accompanying pragma **pack** cannot be split across a package specification and body.

The effect of pragma **pack** is to minimize storage consumption by discrete component types whose ranges permit packing. Use of pragma **pack** does not defeat allocations of alignment storage gaps for some record types. Pragma **pack** does not affect the representations of real types, pre–defined integer types, and access types.

F.1.3 Pragma Suppress

Pragma **suppress** is implemented as described in the LRM section 11.7, with these differences:

- Presently, division_check and overflow_check must be suppressed via a compiler flag, **-fN**; pragma **suppress** is ignored for these two numeric checks.
- The optional "**ON =>**" parameter name notation for pragma **suppress** is ignored.
- The optional second parameter to pragma **suppress** is ignored; the pragma always applies to the entire scope in which it appears.

F.2 Attributes

All attributes described in the LRM Appendix A are supported. The implementation–dependent Meridian attribute **'locoffset** is applied to a parameter and returns as a universal–integer the stack offset of that parameter (the offset from the BP register). It allows machine code insertions to access parameters using less error–prone symbolic names. An example follows.

```
machine_code.inst3'(16#8B#, 16#4E#, nbytes'locoffset);
```

F.3 Standard Types

Additional standard types are defined in Meridian Ada:

- **byte_integer**
- **short_integer**
- **long_integer**

The standard numeric types are defined as:

```
type byte_integer  is range -128 .. 127;
type short_integer is range -32768  ..  32767;
type integer       is range -32768  ..  32767;
type long_integer  is range -2147483648  ..  2147483647;
type float is digits 15
  range -1.79769313486231E+308  ..  1.79769313486231E+308;
type duration is delta 0.0001 range -86400.0000  ..  86400.0000;
```

F.4 Package System

The specification of package **system** is:

```
package system is
  type address is new long_integer;

  type name is (i8086);
  system_name   : constant name := i8086;
```

```
storage_unit    : constant := 8;
memory_size     : constant := 1024;

-- System-Dependent Named Numbers

min_int         : constant := -2147483648;
max_int         : constant := 2147483647;
max_digits      : constant := 15;
max_mantissa    : constant := 31;
fine_delta      : constant := 2.0 ** (-31);
tick            : constant := 1.0 / 18.2;

-- Other System-Dependent Declarations

subtype priority is integer range 1 .. 20;
```

The value of **system.memory_size** is presently meaningless.

F.5 Restrictions on Representation Clauses

F.5.1 Length Clauses

A size specification (**t'size**) is rejected if fewer bits are specified than can accommodate the type. The minimum size of a composite type may be subject to application of pragma **pack**. It is permitted to specify precise sizes for unsigned integer ranges, e.g. 8 for the range **0..255**. However, because of requirements imposed by the Ada language definition, a full 32–bit range of unsigned values, i.e. **0.. (2**32) -1**, cannot be defined, even using a size specification.

The specification of collection size (**t'storage_size**) is evaluated at run–time when the scope of the type to which the length clause applies is entered, and is therefore subject to rejection (via **storage_error**) based on available storage at the time the allocation is made. A collection may include storage used for run–time administration of the collection, and therefore should not be expected to accommodate a specific number of objects. Furthermore, certain classes of objects such as unconstrained discriminant array components of records may be allocated outside a given collection, so a collection may accommodate more objects than might be expected.

The specification of storage for a task activation (**t'storage_size**) is evaluated at run–time when a task to which the length clause applies is activated, and is therefore subject to rejection (via **storage_error**) based on available storage at the time the allocation is made. Storage reserved for a task activation is separate from storage needed for any collections defined within a task body.

The specification of small for a fixed point type (**t'small**) is subject only to restrictions defined in the LRM section 13.2.

F.5.2 Enumeration Representation Clauses

The internal code for the literal of an enumeration type named in an enumeration representation clause must be in the range of **standard.integer**.

The value of an internal code may be obtained by applying an appropriate instantiation of **unchecked_conversion** to an integer type.

F.5.3 Record Representation Clauses

The storage unit offset (the at *static_simple_expression* part) is given in terms of 8–bit storage units and must be even.

A bit position (the **range** part) applied to a discrete type component may be in the range `0..15`, with 0 being the least significant bit of a component. A **range** specification may not specify a size smaller than can accommodate the component. A **range** specification for a component not accommodating bit packing may have a higher upper bound as appropriate (e.g. `0..31` for a discriminant `string` component). Refer to the internal data representation of a given component in determining the component size and assigning offsets.

Components of discrete types for which bit positions are specified may not straddle 16–bit word boundaries.

The value of an alignment clause (the optional **at mod** part) must evaluate to 1, 2, 4, or 8, and may not be smaller than the highest alignment required by any component of the record. On PC–DOS, this means that some records may not have alignment clauses smaller than 2.

F.5.4 Address Clauses

An address clause may be supplied for an object (whether constant or variable) or a task entry, but not for a subprogram, package, or task unit. The meaning of an address clause supplied for a task entry is given in section F.5.5.

An address expression for an object is a 32–bit segmented memory address of type `system.address`.

F.5.5 Interrupts

A task entry's address clause can be used to associate the entry with a PC–DOS interrupt. Values in the range `0..255` are meaningful, and represent the interrupts corresponding to those values.

An interrupt entry may not have any parameters.

F.5.6 Change of Representation

There are no restrictions for changes of representation effected by means of type conversion.

F.6 Implementation–Dependent Components

No names are generated by the implementation to denote implementation–dependent components.

F.7 Unchecked Conversions

There are no restrictions on the use of `unchecked_conversion`. Conversions between objects whose sizes do not conform may result in storage areas with undefined values.

F.8 Input–Output Packages

A summary of the implementation–dependent input–output characteristics is:

- In calls to `open` and `create`, the *form* parameter must be the empty string (the default value).
- More than one internal file can be associated with a single external file for reading only. For writing, only one internal file may be associated with an external file; Do not use `reset` to get around this rule.
- Temporary sequential and direct files are given names. Temporary files are deleted when they are closed.
- File I/O is buffered; text files associated with terminal devices are line–buffered.
- The packages `sequential_io` and `direct_io` cannot be instantiated with unconstrained composite types or record types with discriminants without defaults.

F.9 Source Line and Identifier Lengths

Source lines and identifiers in Ada source programs are presently limited to 200 characters in length.

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

K

L

O

P

Q

R

S

T

U

V

W

X